2009

Investigation of Generalized DSSS Under Multiple Access and Multipath

Indrasena Varakantham
Cleveland State University

INVESTIGATION OF GENERALIZED DSSS UNDER MULTIPLE ACCESS AND MULTIPATH

INDRA SENA REDDY VARAKANTHAM

Bachelor of Electrical Engineering

Jawaharlal Nehru Technological University

May, 2006

Submitted in partial fulfillment of requirements for the degree

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

At the

CLEVELAND STATE UNIVERSITY

May, 2009

This thesis has been approved

for the Department of Electrical and Computer Engineering
and the College of Graduate Studies by

_____

Thesis Committee Chairperson, Dr. Fuqin Xiong

_____

Department/Date

_____

Thesis Advisor, Dr. Murad Hizlan

_____

Department/Date

_____

Dr. Pong Chu

_____

Department/Date

_____

Dr. Nigamanth Sridhar

_____

Department/Date

Acknowledgements

I would like to gratefully acknowledge the enthusiastic supervision of my advisor Dr. Murad Hizlan during this work for his valuable suggestions and guidance through out the thesis. Without his guidance and persistent help this dissertation would not have been possible.

I am grateful to the Electrical and Computer Engineering Department of Cleveland State University and Dr. Fuqin Xiong for providing financial support during my master's program. I appreciate Dr. Fuqin Xiong, Dr. Nigamanth Sridhar and Dr. Pong Chu for being valuable committee members. I am also thankful for all my professors with whom I have taken classes during my master program.

Finally, I am forever indebted to my parents for their understanding, financial support and endless patience when it was most required. I am also grateful for Mr. Madan Venn for his support in completing my thesis documentation.

INVESTIGATION OF GENERALIZED DSSS UNDER MULTIPLE ACCESS AND MULTI PATH

INDRA SENA REDDY VARAKANTHAM

ABSTRACT

In this thesis we investigate and compare the average performances of ordinary and generalized direct sequence spread spectrum (DSSS) systems under multi-path fading and multiple-access interference. As part of multiple access performance, we also consider generation of orthogonal and semi-orthogonal codes using various algorithms, and compare cross correlation properties of codes formed by 2-level and 3-level signature sequences.

In order to simulate ordinary and generalized DSSS performance under various scenarios, we develop a complete Java library with classes that are well encapsulated with regard to communication modules and loosely coupled so that we can reuse them to create any type of DSSS communication model. We verify the library under Gaussian noise before performing simulations under multi-path fading and multiple-access interference.

We find with regards to multi-path fading that generalized DSSS does not perform any better than ordinary DSSS, regardless of how the signature sequences are generated. For multiple access, when using perfectly orthogonal signature sequences, we observe that ordinary and generalized DSSS perform exactly the same. We investigate semi-orthogonal sequences in great detail, and observe that generalized DSSS can accommodate more users than ordinary DSSS for the same performance.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Background

This chapter covers the basic concepts of spread spectrum, generalized direct sequence spread spectrum, multiple access, multi-path effect and basics of orthogonal signals.

1.1.1 Spread Spectrum

A class of modulation techniques called “Spread Spectrum,” which was originally used in military applications, has been gaining widespread commercial applications over the past two decades. Spread spectrum mechanism can be combined with multiple access methods to create Code Division Multiple Access (CDMA). Spread spectrum signal is defined as a signal which uses significantly more bandwidth than the message base band signal to transmit. There are two main properties for spread spectrum, one is that the transmitted signal bandwidth is much greater than the base band signal and the other is

that the carrier signal must be in a particular format so that it can be used to decode the original signal at the receiver end. Spread spectrum is based on the concept of spreading the base band signal with the help of fast codes which are called spreading codes. These spreading codes are used in the receiver again to demodulate the original signals. The spreading codes have different properties and they have wide band signals and noise like signals. Spreading by using these kinds of signals has many advantages as these signals are anti-jamming, difficult to be detected by a third party, more tolerant to arbitrary noise, and less affected by multi-path and multiple access interference [1]. There are two main types of spread spectrum:

- Direct Sequence Spread Spectrum (DSSS).
- Frequency Hopping Spread Spectrum (FHSS).

Direct Sequence Spread Spectrum uses a carrier that has a fixed frequency. In DSSS the stream of information is multiplied by the pseudo noise (PN) sequences before transmission and the same process is done to de-spread the sequence at the receiving end. The PN sequence used in DSSS has only two bi polar signals as shown in Figure 1.

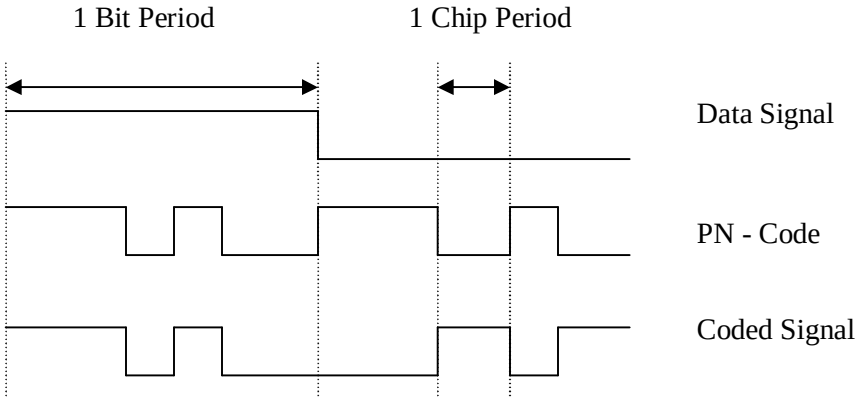

Figure 1 : Direct Sequence Spread Spectrum signaling

Frequency Hopping Spread Spectrum (FHSS) can be exactly termed as "multiple frequency, code selected, frequency shift keying (FSK)". The main principle of FHSS is similar to FSK except that FSK uses only two frequencies, f_1 for positive or "1" and f_2 for negative or "-1". On the other end FHSS uses a large number of frequencies, "usually 2^{20} discrete frequencies". The frequencies are selected on the basis of code word and data signal. In the Figure 2 f_1, f_2 and f_3 are selected on the basis of a PN-code and data signal. The number of frequencies used in the FHSS depends on the requirements of the system, available bandwidth and complexity of the system.

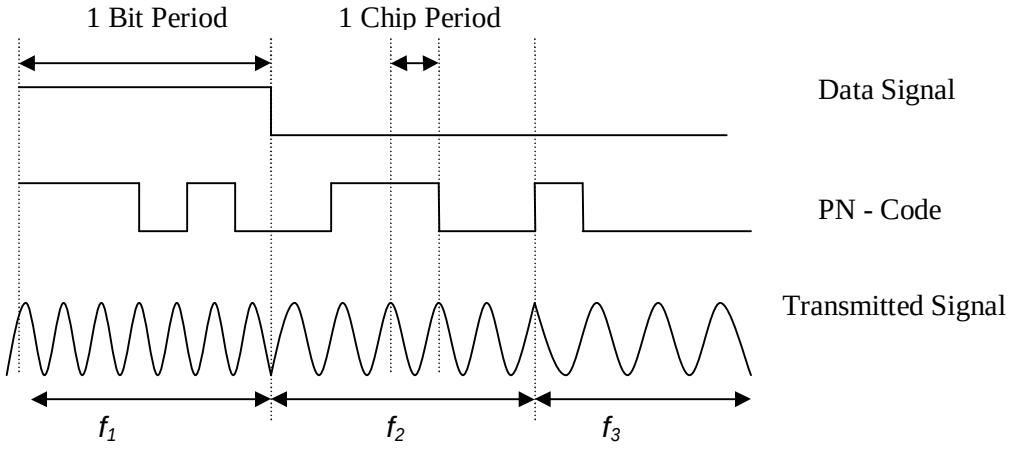

Figure 2: Frequency Hopping Spread Spectrum signaling

1.1.2 Generalized Direct Sequence Spread Spectrum

Generalized Direct Sequence Spread Spectrum (GDSSS) described in [2] is a direct sequence spread spectrum system where PN sequence used has three levels $\{-1, 0,$

1} instead of two levels $\{-1, 1\}$. To differentiate between the two cases we call the two-level direct sequence spread spectrum "ordinary direct sequence spread spectrum" and three-level, "generalized direct sequence spread spectrum." As previously shown in [2] the ordinary spreading sequence is a special class of modulation schemes called random modulation which becomes asymptotically optimal as the number of signal dimensions *N* approach to infinity when the message symbol is uniformly distributed on the surface of an *N*-dimensional sphere.

The signal representation for the sample code word 10110100 is shown in Figure 3 where 1 is represented by {+1} amplitude and 0 is represented by $\{-1\}$ amplitude. For 2-level code the possible amplitudes are only two which are shown in Figure 3.

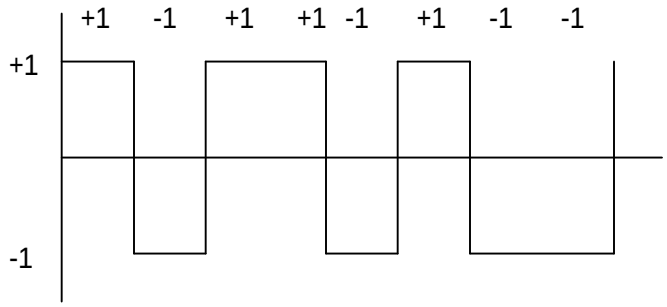

Figure 3: Signal representation of code word for 2-level

In Figure 4 all the possible transmitted symbols of ordinary DSSS is shown. All the possible vectors occupy the vertices of a cube. Message signals are randomly distributed on the vertices of a 3-dimensional cube if the code length is 3. So as shown in Figure 4 the maximum possible codes formed for a length of three is 8 (2^n where *n* is length of code word, i.e. 3 in this case) which are all possible vertices of a cube.

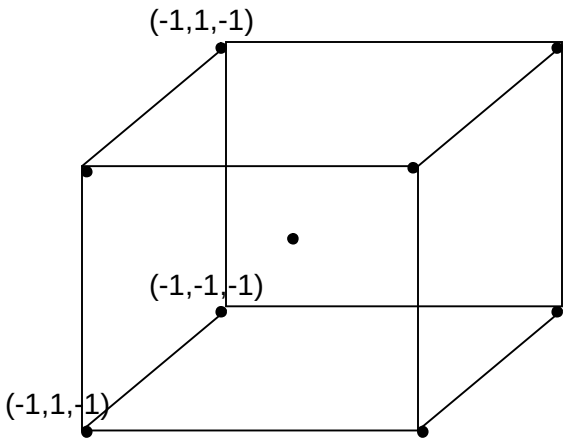

Figure 4: Vectors of ordinary DSSS

Generalized direct sequence spread spectrum can be viewed as the use of 0 in the chip sequence in addition to $\{1,-1\}$. In Figure 5 a signal representation with $\{+1,\ 0,-1\}$ is shown for a code word of 12010122 where 1 is transmitted by +1 and 0 is transmitted by 0 and 2 is transmitted by -1.

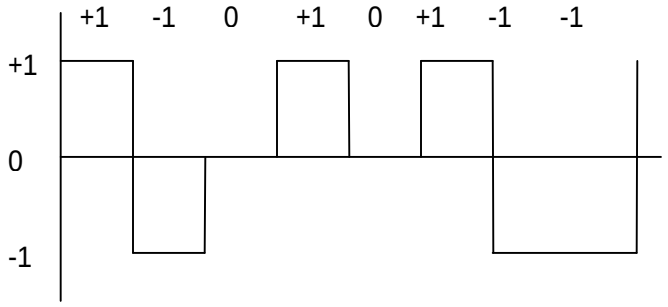

Figure 5: Signal representation of code word for 3-level without energy normalization

In Figure 6 all the possible vectors of code length 3 are formed by 3-level code word is shown on a cube. As the energy is not normalized all the vectors do not have equal energy and they lie on the cube. The black circles have more energy because of the

absence of zeros and would lie on the vertices. The white circles have less energy because of the zeros present in the sequence.

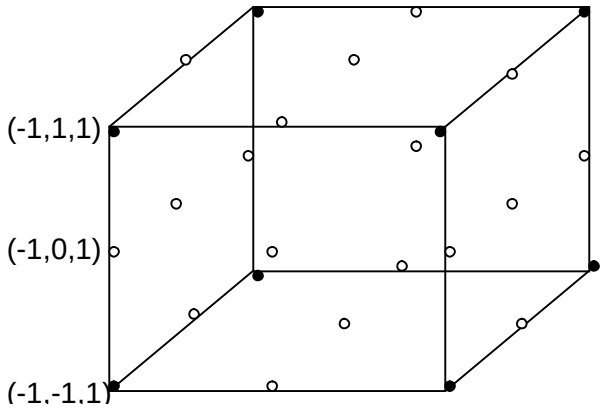

Figure 6: Transmission vectors for generalized DSSS over a cube

As the number of zeros increase in the sequence, the energy to carry the information decreases, this makes the signal more error prone to the noise. So to keep the same energy as in 2-level the loss of energy due to zeros is added to +1 or -1 amplitudes of the sequences. The energy is normalized by the given formula:

$$Normalizat\ ion\ Factor = \sqrt{\frac{n}{u}}$$

where n *is total number of bits*

u *is total number of* $1's$ *present in sequence*

The normalized signal is shown in Figure 7 where the amplitudes for +1 and -1 are increased by the "normalization factor". In Figure 8 all the vectors are energy normalized and plotted on a 3-dimensional space. This forms a sphere where the code

vectors lie on the surface of the sphere. All the possible codes formed have equal energy and maximum number of code words formed by 3-level is much larger i.e. 27 (3^n where n is length of code word, i.e. 3). Say if the normalization factor is 1.2 then the signal would be as shown in Figure 7.

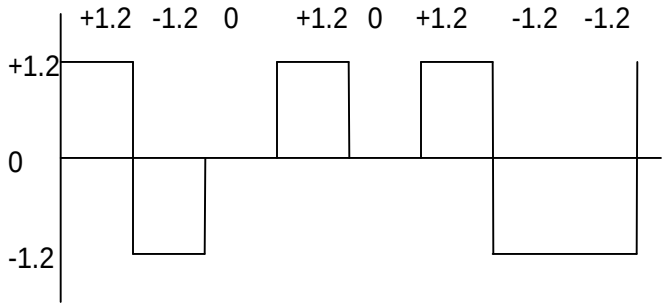

Figure 7: Signal representation of 3-level with energy normalization

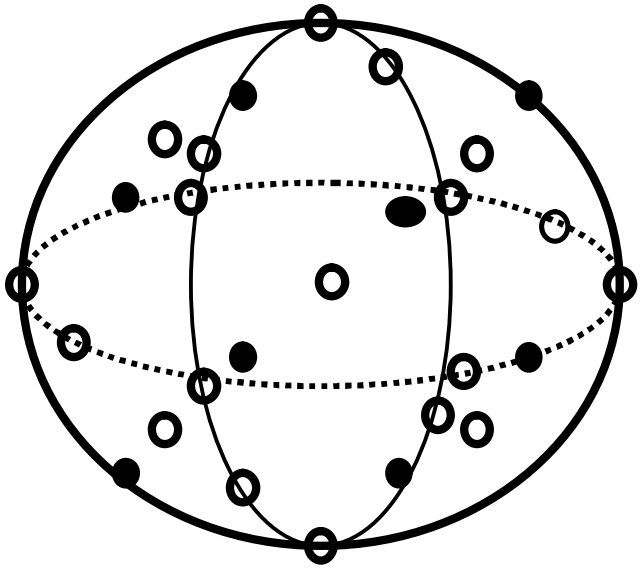

Figure 8: Transmission vector for Generalized DSSS over a sphere

1.1.3 Multiple-Access

Multiple-Access is a technique where many users can access the same channel simultaneously. There are several different ways to access channel. The most important access techniques are shown below.

1) Time Division Multiple Access (TDMA).

2) Frequency Division Multiple Access (FDMA).

3) Code Division Multiple Access (CDMA).

In Time Division Multiple Access all users transmit with same frequency but in different time slots. Every user transmits the signal in its own time slot and will be idle for rest of the period. For example in TDMA if user 1 is allocated with time slot 1 then user 1 only transmits in time slot 1 and remains idle for rest of the time and if user 2 is allocated with time slot 2 then user 2 transmits in time slot2 and remains idle for rest of the time. This makes user 1 communication not to interfere with user 2 and vice versa.

In Frequency Division Multiple Access (FDMA) all users transmits at the same time but with different frequencies. The total available bandwidth is divided into different frequency bands and assigned to different users. Every user transmits the signal with frequency allocated for it.

In Code Division Multiple Access (CDMA) every user transmits the signal at the same time and with same frequency but modulated with different spreading codes. User separation at the receiver is possible because each user spreads the modulated waveform over a wide bandwidth using unique spreading codes. CDMA generally has two types of spreading principle like DSSS (Direct Sequence Spread Spectrum) and FHSS (Frequency

Hopping Spread Spectrum). In this thesis we are going to see DSSS. This is briefly explained in Section 4.1.3 Code Division Multiple Access.

1.1.4 Multi-Path Fading

Multi-path is a simple term used to describe that multiple paths of a radio wave can propagate between transmitter and receiver. This propagation includes ground wave, reflection from earth, ionosphere reflection, reflection from the buildings and so on. The reflected waves have different time delays and different path losses depending on the reflected surface. These multi-path signals add vectorially and produce fluctuations in the received signal known as **fading**. The effects of multi-path fading include signal shifting, construction and destruction of the signal. In digital communication this effect can cause errors and affect the quality of the communication. The signal model and multi-path fading is further explained in Section 3.1 Introduction. There are many techniques used to minimize the effects of fading in digital communication. In spread spectrum systems m-sequences are used as signature sequences to eliminate the multi-path effects on the systems.

1.2 Motivation

The motivation for this thesis is mainly based on three points.

1) For worst-case noise scenario generalized spread spectrum is more efficient than ordinary spread spectrum as stated in [3].

2) Auto correlation properties for m-sequences generated for 3-level is better than 2-level as stated in [4].

3) For a particular code length number of possible codes formed for 3-level (3^n where n is length of the code word) is much greater than 2-level (2^n where n is length of the code word).

As stated in [2], [3] and [5] for worst-case scenario generalized (N-level) spread spectrum is more efficient than ordinary (2-level) spread spectrum. In [6] and [7] it is observed that generalized spread spectrum is still better when using coding techniques like block code and convolution codes. But in all cases research was done on worst case error scenario which helps for a more robust communication system, but does not investigate on practical scenarios like multi-path scenario, multiple-access and Gaussian noise scenario. This motivated our research to see the effects on other scenarios with interference like multi-path and multiple-access and compare the performance for 2-level and 3-level.

In [4] the author proposed that there is way of generating m-sequence for generalized signature sequence like m-sequence. It also showed that generalized m-sequences improve a lot on the auto correlation properties than ordinary sequences. In the thesis they showed that the auto correlation properties take a three valued function $(1, 0, -1)$. This gives a possibility of researching the effects on the system with multi-path fading scenario hoping to improve the performance as the auto correlation properties for generalized spread spectrum is zero compared with $\left(\frac{-1}{N}\right)$ where N is the sequence length for ordinary spread spectrum (2-level). As shown in Figure 25, if the reflected path delay is greater than one chip then the energy from the reflected signal is zero.

We can observe in Figure 8 that the maximum possible codes for 3-level (3^n where n is length) is much greater than 2-level (2^n where n is length). This creates the possibility of having more semi-orthogonal or orthogonal codes for 3-level then 2-level which can be used in multiple-access. As stated in [8] Gold sequences which are used in multiple-access are generated using m-sequences. This motivated us to generate Gold sequences for generalized spread spectrum using m-sequences for 3-level, and compare the cross correlation properties with 2-level sequences. The generated Gold sequences are used as codes in multiple-access and in comparing the bit error rate (BER) results between ordinary and generalized spread spectrum.

All the above claims had inspired to investigate the simulation in multiple-access interference and multi-path interference and observe the difference between the ordinary DSSS and generalized DSSS. The simulation is performed under the practical conditions and taking most of interference into consideration. The simulation uses the general conditions of IS-95 [8] model and BER response for ordinary and generalized DSSS are observed.

1.3 Related Work

In [5] the authors proposed and asymptotically optimal modem which distributes a codeword uniformly on an *N*-dimensional sphere. In [5] the authors show that ordinary DSSS is a special case of, random modulation and that random distribution on the surface of an *N*-dimensional sphere becomes asymptotically optimal as *N* gets larger, minimizing the signal-to-interference ratio (SIR) for a given worst case performance level. In [2], ordinary DSSS has been generalized so that the possible transmitted vectors have been

selected to have one of the vectors shown in Figure 5 and radially project around a 3-D sphere as shown in Figure 8. The author in [3] has described the performance of coded ordinary DSSS in arbitrary noise with an upper bound on worst case error probability which is incurred by the communication system and observed that with coding, there is much to gain in going from ordinary DSSS to asymptotically optimal random modulation. The author in [9] considered further generalization of uncoded ordinary and generalized DSSS and observed generalized DSSS performs better than ordinary DSSS.

In [6], [7] and [10] the authors has proved that generalized spread spectrum has relatively better performance compared to ordinary spread spectrum when using coding techniques. In [10], coded and interleaved generalized direct sequence spread spectrum is considered and observed the improvement in performance for generalized DSSS than ordinary DSSS. The author in [6] described coded ordinary and coded generalized spread spectrum with various cyclic, BCH and burst error correcting codes and proved that the worst case performance for generalized DSSS is better than ordinary DSSS. In [7] the thesis has observed performance for generalized DSSS and ordinary DSSS with convolutional codes with various constraint lengths, code lengths and observed that generalized DSSS consistently performs better than ordinary DSSS.

The author in [4] has worked on generation of 3-level and 5-level m-sequences and observed the auto correlation properties. In [4] the author showed software and hardware implementation for generation of m-sequences for 3-level and 5-level and observed the auto correlation and randomness properties of such sequences. The auto correlation properties suggest that the PN sequence produced for generalized sequences have good randomness than ordinary PN sequences. It has FPGA implementation for

generating m-sequences and can be used for practical modulation in DSSS without much complex circuits. In [11] the thesis has observed the performance for generalized DSSS is better than ordinary DSSS for worst case multipath interference.

In [8], it is shown that for generating 2-level Gold sequences, two sets of m-sequences which are generated by Linear Feedback Shift Registers (LFSR) are used. The cross correlation properties of Gold sequences have a constant angle and approximately equal to 90^0 (not exactly 90^0), which are semi-orthogonal codes used for multiple access to separate the users. In this thesis we have combined the results from [4] and procedure from [8] for generating 3-level Gold Sequences using LFSR. To have accurate simulation results we have used delay spread explained in [12] and used them to simulate in multi-path scenario.

1.4 Thesis Structure

The thesis starts with the introduction of the simulation environment, which is used to simulate the entire scenario. In the Chapter 2 there is a short description of the environment used for the simulation and description of some library functions which are used in the thesis. This Chapter has the proof and Gaussian example of the designed library. In Chapter 3 we investigate the results of generalized DSSS and ordinary DSSS for multi-path with random codes and m-sequences as signature sequences. Then we move on to Chapter 4 where we see the scenario and simulation results of multiple-access scenario. In this chapter we see all the scenarios of multiple-access like random code, orthogonal codes and semi-orthogonal codes. In Chapter 5 we see the methods to

generate orthogonal codes which have pre determined algorithm and search based algorithm. This chapter also provides analysis of the codes for higher code length. Finally we come to conclusion and future work.

CHAPTER II

SIMULATION

As simulation of the communication system involves a lot of computation there is requirement of developing a model specific program. Java programming language is used to simulate all the scenarios in this thesis. The choice of java programming is due to its object-oriented nature and that it can be used and extended easily by other researchers.

2.1 Explanation of Simulator

A block diagram for the simulator is shown in Figure 9 where each block is a component in Java which can be used for other scenarios as well. All the modules are briefly explained in the Section 2.3. The simulator can be clearly explained by the flow of the signal through out the system. A Gaussian simulation is shown for the system, which uses all the basic components of the provided library.

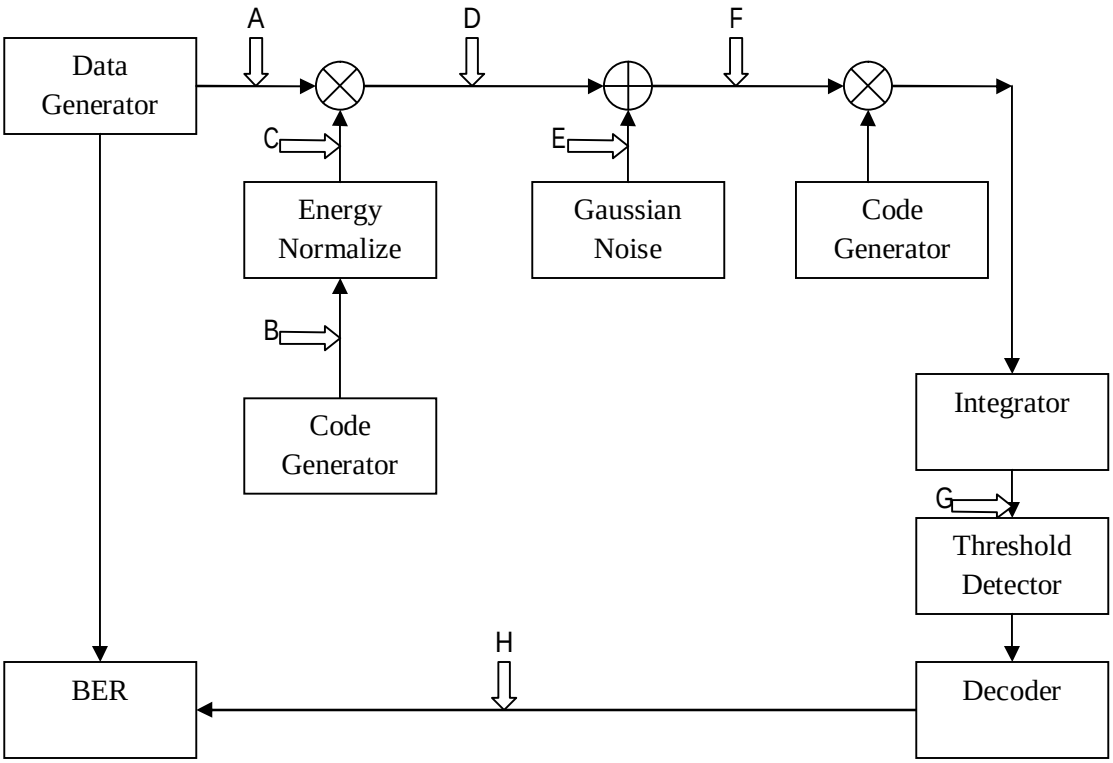

Figure 9: Block Diagram of Gaussian Noise scenario for generalized DSSS

2.1.1 Ordinary Spread Spectrum Simulation

Data Generator:

java.util.Random is used to produce the data sequence which is random and depends upon the seed given to the class. The output values are +1 and −1 stored in an integer array where the occurrence of both +1 and −1 has equal probability. For the same seed the random sequence produced by this generator is same. For this module we have used current system time in milli seconds as seed which helps to generate different output for every simulation making the simulation results more accurate. Sample output is shown at point A in Section 2.3.

Code Generator:

It produces a unique code for entire modulation and demodulation depending on the seed given to the class. The codes generated by the module depend on the seeds given to the module. The same seed is given to the random generator at receiver to have the same codes. The possible levels produced are $+1$ or -1.

Multiplier:

This module is used to multiply the code word and the data sequence. The output is the sequence resulted in multiplication of two arrays. The length of bits produced is data length × code length.

Noise Generator:

The noise generator produces random double values, which are used as noise for the entire system. For the above block diagram the random values produced by the module have Gaussian distribution. The standard deviation is 1 and the mean is zero.

Adder:

This module is used to add noise signal and data signal, which are used in the communication channel. This acts as the channel where the noise is added to the signal.

Integrator:

This module integrates the given signal and produces the integrated value over a period (over code length) and produces the output. The integrated value can be seen at point G.

Threshold detector:

This module compares the integrated output from previous module with a threshold value and decides the input bit transmitted. The integrated value for ordinary spread spectrum is compared with a value 0.

$$Output = \begin{cases} -1 \; if \;\; input < 0 \\ +1 \; if \; input \; > 0 \\ \pm 1 \; if \; input = 0 \end{cases}$$

BER:

It compares the decoded data with the original or transmitted data and stores the total number of error bits and after reaching specified number of errors or specified number of bits it calculates bit error rate by the given formula:

$$BER \;\; = \;\; \frac{number\ of\ errors\ accumulated}{number\ of\ bits\ accumulated}$$

2.1.2 Generalized Spread Spectrum Simulation

All the modules are same for both ordinary spread spectrum and generalized spread spectrum except few modules like Code Generator, generalization of spectrum and threshold detector. This section describes only the modules which are different from ordinary spread spectrum.

Code Generator:

The possible levels in this simulation are (-1, 0, +1). Sample output is shown at point B in Section 2.3.

Generalization of Spectrum:

Energy for 3-level and 2-level are not same due to the loss in zero. To compensate for the energy loss of zeros the sequence is multiplied by normalization factor, so the energy is equal for both generalized and ordinary sequence.

$$Normalization\ Factor = \sqrt{\frac{n}{u}}$$

$$where\ n\ is\ total\ number\ of\ bits$$

$$u\ is\ total\ number\ of\ 1's\ present\ in\ sequence$$

Sample output is shown at point C in Section 2.3.

Threshold Detector:

It is used as a comparator which compares the integrated value with a threshold and generates output by following threshold.

$$Output = \begin{cases} -1 & if\ input < -0.5 \\ -1\ or\ 0 & if\ input = -0.5 \\ 0 & if\ -0.5 > input > 0.5 \\ 0\ or\ +1 & if\ input = 0.5 \\ +1 & if\ input > 0.5 \end{cases}$$

2.2 Algorithm and Flow Chart

Algorithm:

1) Accept Code Length, Data Length and SNR.
2) Initialize data array and code array.
3) Multiply data array and code array.
4) Initialize Noise Array.
5) Calculate Gaussian Noise and store in noise array.
6) Add noise array and resultant multiplied array.
7) Multiply the resultant array again with code.
8) Integrate the resultant array.
9) Decide the bits. If the resultant array has >0 decide the bit has 1 and <0 the bit is -1.
10) Calculate the errors.
11) If the errors are less than 100 then go to 2.
12) Increase snr by 1 and calculate BER.
13) If snr< user given max snr then go to 1.
14) Print BER vs SNR.
15) End.

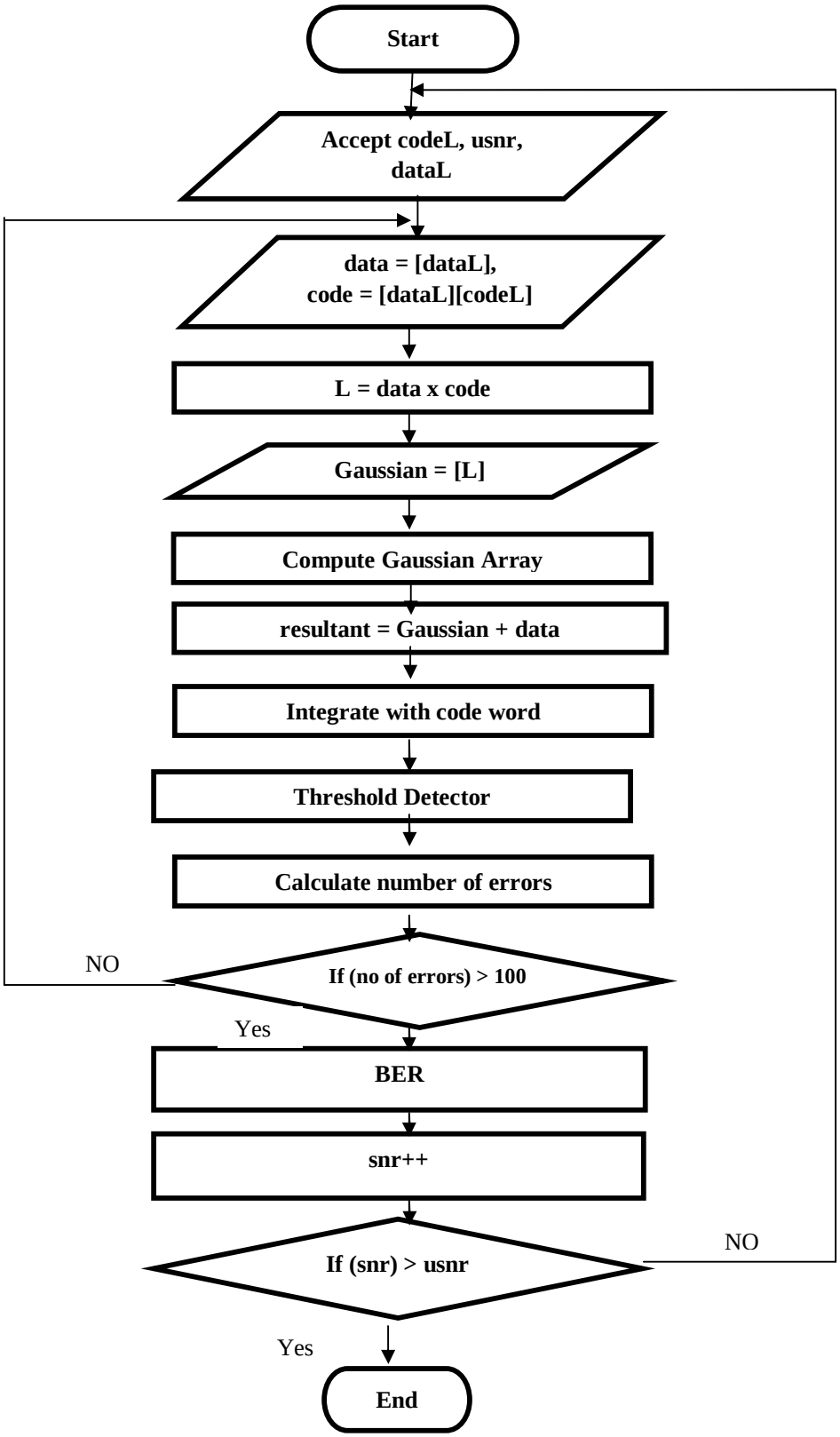

Figure 10: Flow Chart of the simulation

2.3 Flow of the Signal in Simulator

Signal starts from PN sequence generator which produces an integer array (point A) with length of user specified length. The array circulates around the system and compares with decoded array at the receiver side. All the manipulation takes place on the array which PN Sequence modulator has produced.

Point A: Data Sequence

```
[-1,   1,    -1,    1,    -1,    1,    1,    1,    -1,    -1]
```

This signal produced by the PN Generator and is multiplied by the code generated from the code generator. The code generator takes the code length, number of levels and data length as arguments and generates a 2-dimensional integer or double array depending on the number of levels. For 2-level and 3-level the generated array has a possible integer value of 1, 0 or -1. A sample output is shown at point B. The 2-dimensional arrays were multiplied by the signal array and produce another 2-dimensional array. The multiplication of the array is done by calculating the first element in the integer array to first row in the code array. Thus the total signal is multiplied by the code. The multiplied 2-dimensional array is shown at point C.

Point B: Code Word

$$\begin{pmatrix} 0 & -1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ -1 & 1 & -1 & 0 & -1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 1 & 1 & -1 & 1 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \\ -1 & 1 & 1 & 1 & 1 & -1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & -1 & 1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 1 & 1 & -1 & 1 & 1 & 1 & -1 \\ 1 & -1 & 0 & -1 & 0 & -1 & 1 & -1 & 1 & -1 \\ -1 & 0 & 0 & 0 & -1 & -1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & -1 & 0 & 0 & 1 \end{pmatrix}$$

Point C: Generalized Sequence = code sequence × normalized power factor

$$\begin{pmatrix} 0.0 & -1.82 & 0.0 & 0.0 & 0.0 & 0.0 & 1.825 & 1.825 & 0.0 & 0.0 \\ -1.29 & 1.290 & -1.29 & 0.0 & -1.29 & 0.0 & 1.290 & 0.0 & 1.290 & 0.0 \\ 0.0 & 0.0 & 1.290 & 0.0 & 0.0 & -1.29 & 1.290 & 1.290 & -1.29 & 1.290 \\ -1.82 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -1.82 & 0.0 & 1.825 \\ -1.19 & 1.195 & 1.195 & 1.195 & 1.195 & -1.19 & 1.195 & 0.0 & 0.0 & 0.0 \\ 1.195 & 1.195 & 1.195 & 0.0 & 0.0 & -1.19 & 1.195 & -1.19 & 0.0 & -1.19 \\ -1.0 & -1.0 & -1.0 & 1.0 & 1.0 & -1.0 & 1.0 & 1.0 & 1.0 & -1.0 \\ 1.118 & -1.11 & 0.0 & -1.11 & 0.0 & -1.11 & 1.118 & -1.11 & 1.118 & -1.11 \\ -1.41 & 0.0 & 0.0 & 0.0 & -1.41 & -1.41 & 1.414 & 0.0 & 1.414 & 0.0 \\ 1.414 & 0.0 & 0.0 & 1.414 & 1.414 & 0.0 & -1.41 & 0.0 & 0.0 & 1.414 \end{pmatrix}$$

Point D: Multiplied Sequence = Generalized sequence × data sequence

$$\begin{pmatrix} -0.0 & 1.825 & -0.0 & -0.0 & -0.0 & -0.0 & -1.82 & -1.82 & -0.0 & -0.0 \\ -1.29 & 1.290 & -1.29 & 0.0 & -1.29 & 0.0 & 1.290 & 0.0 & 1.290 & 0.0 \\ -0.0 & -0.0 & -1.29 & -0.0 & -0.0 & 1.290 & -1.29 & -1.29 & 1.290 & -1.29 \\ -1.82 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -1.82 & 0.0 & 1.825 \\ 1.195 & -1.19 & -1.19 & -1.19 & -1.19 & 1.195 & -1.19 & -0.0 & -0.0 & -0.0 \\ 1.195 & 1.195 & 1.195 & 0.0 & 0.0 & -1.19 & 1.195 & -1.19 & 0.0 & -1.19 \\ -1.0 & -1.0 & -1.0 & 1.0 & 1.0 & -1.0 & 1.0 & 1.0 & 1.0 & -1.0 \\ 1.118 & -1.11 & 0.0 & -1.11 & 0.0 & -1.11 & 1.118 & -1.11 & 1.118 & -1.11 \\ 1.414 & -0.0 & -0.0 & -0.0 & 1.414 & 1.414 & -1.41 & -0.0 & -1.41 & -0.0 \\ -1.41 & -0.0 & -0.0 & -1.41 & -1.41 & -0.0 & 1.414 & -0.0 & -0.0 & -1.41 \end{pmatrix}$$

Next the channel where Gaussian noise exists is an ideal channel. This ideal channel means that it does not have any filtering effects and passes all frequencies. This

makes the system deal only with the arrays produced by the PN sequence generator. As there are no filtering effects there is no need of having extra samples for each bit as each bit can pass through the signal without any effect. This type of communication model makes the simulation much faster than using communication simulators existing in the market.

Next the signal is added to a Gaussian noise which is generated by Gaussian module. The Gaussian noise generator generates a double array of length $N * L$ with the built-in Gaussian function provided by Java library. This generated array is added to 2-dimensional resultant arrays. Next this signal is given to the receiver side where demodulation process starts. The receiving module contains multiplier, integrator and threshold detector.

Point E: Noise Sequence

-1.50	-1.97	-1.22	-3.72	3.508	0.626	-1.18	1.731	-4.13	-2.92
-3.51	0.866	-2.42	-1.89	-0.83	0.455	-0.82	1.614	-1.65	-4.13
-1.31	-0.92	0.875	-2.24	-1.99	-0.67	-3.87	-1.21	0.259	2.464
-0.11	1.639	-3.63	-1.15	-1.30	-0.57	-0.48	0.370	-3.17	-3.96
0.441	-2.34	0.143	0.766	1.142	-0.91	-1.61	2.423	-2.15	1.118
-2.59	1.438	0.638	-0.00	-2.70	0.197	0.527	-1.64	3.364	-0.48
-1.72	0.904	0.680	-0.47	-1.29	-4.50	0.085	-3.49	0.143	-1.87
-1.28	2.332	1.311	-2.52	-1.94	2.262	-2.24	-2.22	-0.48	0.131
0.892	3.969	1.228	0.467	-2.40	-2.16	-1.62	2.297	3.583	0.521
1.991	3.876	-0.55	4.594	1.320	1.021	-4.30	0.362	7.937	0.367

Point F: Noise plus sequence = noise sequence + multiplied

-1.50	-0.14	-1.22	-3.72	3.508	0.626	-3.01	-0.09	-4.13	-2.92
-4.80	2.157	-3.71	-1.89	-2.12	0.455	0.469	1.614	-0.35	-4.13
-1.31	-0.92	-0.41	-2.24	-1.99	0.620	-5.16	-2.50	1.550	1.173
-1.94	1.639	-3.63	-1.15	-1.30	-0.57	-0.48	-1.45	-3.17	-2.13
1.637	-3.54	-1.05	-0.42	-0.05	0.281	-2.81	2.423	-2.15	1.118
-1.39	2.633	1.833	-0.00	-2.70	-0.99	1.722	-2.83	3.364	-1.68
-2.72	-0.09	-0.31	0.522	-0.29	-5.50	1.085	-2.49	1.143	-2.87
-0.16	1.214	1.311	-3.64	-1.94	1.144	-1.12	-3.34	0.629	-0.98
2.306	3.969	1.228	0.467	-0.99	-0.75	-3.03	2.297	2.168	0.521
0.577	3.876	-0.55	3.180	-0.09	1.021	-2.89	0.362	7.937	-1.04

At the receiving end the first step is to multiply with the code word, which is similar to correlating the signal with the code word. This signal is fed to integrator where it integrates and produces the correlation energy of the signal. In integrator first it adds all the columns in a row on resultant array and divides it by the column length. This resultant output is stored in a new array, which is correlated energy between the signal and code. Now we have a double array with length exactly equal to the original data length. This resultant array is given to the threshold detector. For the threshold detector we have used a hard decision, which decides the bit as 1 if the correlated energy is above zero, or 0 if the correlated energy is below zero. The decoded bit array is compared with the original bit array in the BER module. The BER calculates the number of errors for number of bits occurred.

Point G: Multiplied and integrate sequence

[-5.40	16.66	-11.7	2.303	-11.7	12.32	11.49	5.530	-2.01	7.7]

Point H: Decoded Sequence

```
[-1,   1,    -1,    1,    -1,    1,    1,    1,    -1,    1]
```

2.4 Validity of Simulation Results

As the simulation is based on the Monte Carlo loops the above process repeats until it reaches a predefined number of errors (error count) or predefined number of bits (bit count). The number of loops depends upon the confidence level required. Error counting is a method of counting the bits until a predefined number of errors have been reached and bit counting is a method of counting the errors until a predefined number of bits have been reached. The error or bit count value depends on the confidence level required. For 100% confidence level it requires infinite loops to be executed; as it is impossible to attain infinite loops the simulation is done for bit count where the number of errors is fixed. In this thesis, we use error counting suche that whenever the number of errors reaches 100, the systems exits from the loop and computes the BER.

The value of number of errors to be accumulated depends upon two factors, *confidence level* and *confidence interval*.

Confidence Interval: It is a plus or minus range for simulated BER where the actual value can be present. The end points of confidence level are reffered to as confidence limits.

Confidence Level: It determines the level of certainity that the actual value lies in the specified confidence interval. It is expressed as a percentage like 95% or 99%.

Number of errors, number of trials, confidence level and confidence interval are interdependent. Let N be the number of trials, n be the number of errors, C be the confidence level, BER be the estimated error probability and p be the actual error probability. Then, from [13]

$$\Pr[x_+ \le p \le x_-] = \frac{C}{100} \qquad \text{(a)}$$

where

$$x_\pm = \frac{n}{N}\left[1 + \frac{d_\beta{}^2}{2n}\left(1 \pm \sqrt{\frac{4n}{d_\beta{}^2} + 1}\right)\right], \qquad \text{(b)}$$

$$x_+ = BER \times a \ , \quad x_- = \frac{BER}{a} \ , \quad a > 1 \ , \qquad \text{(c)}$$

$$BER = \frac{number\ of\ errors\ accumulated}{number\ of\ bits\ accumulated} = \frac{n}{N} \qquad \text{(d)}$$

and d_β is given by

$$\frac{1}{\sqrt{2\pi}} \int_{-d\beta}^{d_\beta} e^{-t^2/2} dt = \frac{C}{100} \ . \qquad \text{(e)}$$

Clearly, the lower the value of a, the tighter the confidence interval will be. In this thesis, we consider a 99% confidence level and keep the number of errors accumulated fixed at $n = 100$. From (e), with $C = 99$, we get d_β as follows:

$$\frac{1}{\sqrt{2\pi}} \int_{-d\beta}^{d_\beta} e^{-t^2/2} dt = 0.99$$

$$erf\left(\frac{d_\beta}{\sqrt{2}}\right) = 0.99$$

From the error function table we get

$$\frac{d_\beta}{\sqrt{2}}=1.824$$

$d_\beta = 2.58$

Now, from (b), (c) and (d), we get

$$BER \times a = BER\left[1+\frac{{d_\beta}^2}{2n}\left(1+\sqrt{\frac{4n}{{d_\beta}^2}+1}\right)\right]$$

$$a=\left[1+\frac{{d_\beta}^2}{2n}\left(1+\sqrt{\frac{4n}{{d_\beta}^2}+1}\right)\right] \qquad \text{(f)}$$

Finally, by substituting all the values in (f), we get

$$a=\left[1+\frac{2.58^2}{2\times 100}\left(1+\sqrt{\frac{4\times 100}{2.58^2}+1}\right)\right]$$

$$a=1.293$$

We therefore have a confidence interval of $\left[\frac{BER}{1.293} < p < BER \times 1.293\right]$ for all error probability estimates in this thesis, i.e. $\Pr\left[\frac{BER}{1.293} < p < BER \times 1.293\right] = 0.99$. In other words, we are 99% sure that our BER estimate will be no more than 29.3% higher than the true value of the error probability, and the true value of the error probability will be no more than 29.3% higher than our BER estimate.

2.5 Simulation Results

The output of the ordinary DSSS and generalized DSSS is shown in Figure 11. The output is compared with BPSK as theoretically both the outputs should match with BPSK in the presence of Gaussian noise only. If we observe the output, three of them perform the same, which is the proof for the designed simulator to be working correctly [14].

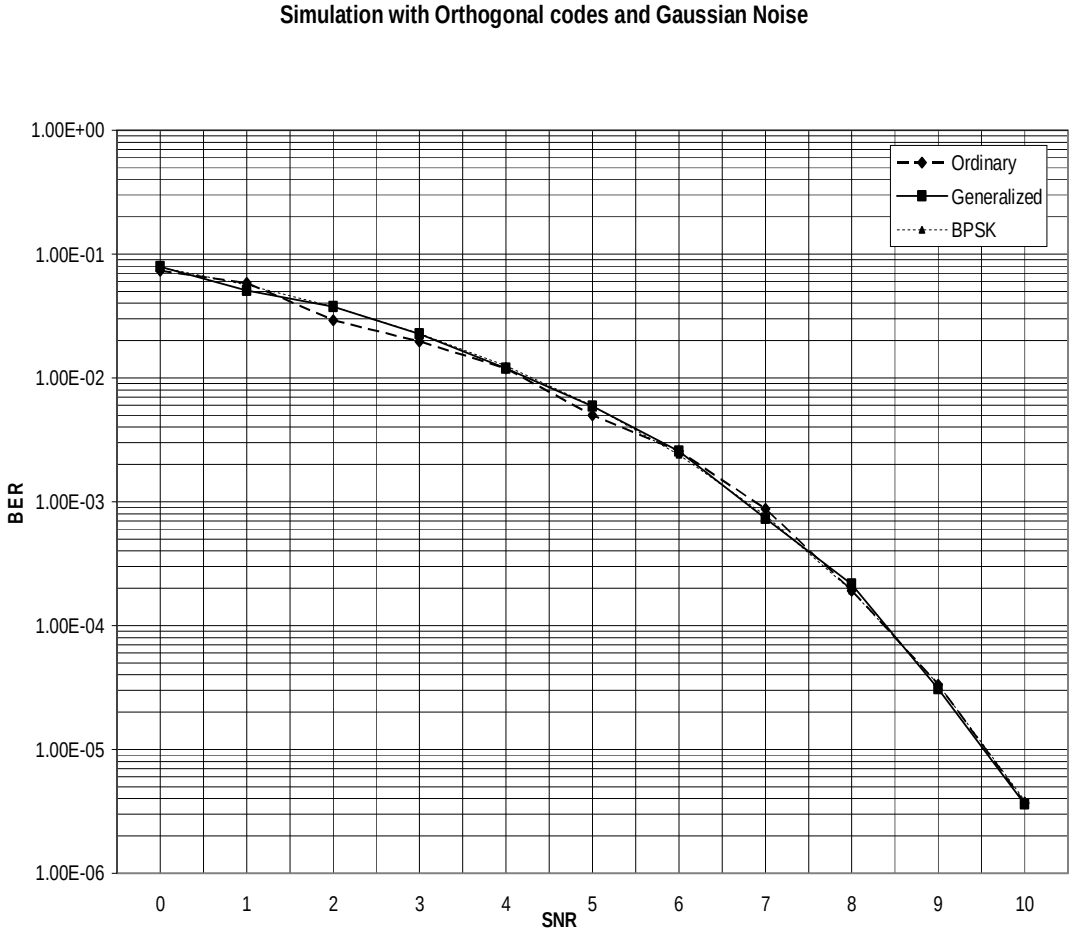

Figure 11: Comparison between ordinary DSSS, generalized DSSS and BPSK under Gaussian noise

2.6 Simulator Provided Library Function

As a part of this thesis we have developed a lot of useful functions which can be used by other researchers to simulate there own scenarios. The simulator library provides a large set of library functions for generating and manipulating the signal. This library has java version programs of existing research like generation of m-sequence equations for generalized sequence, generation of Gold sequence, etc. This library has sample simulation programs for Gaussian, multi-path and multiple Access environments which can be used as a reference for other scenarios. Main packages in the project are application, coder, generator, interleaver, and util. All the classes for every package are explained below.

Package: dsss.util

This package contains all the utilities to manipulate the signal. The main components in this package are Multiplier, Correllator, Integrator, Print, Quantizer, Sampler, Shifter, Addition, BER, and Converter.

Fully qualified class name: *dsss*.util.Multiplier

Multiplier: This class is responsible for multiplying any kind of signal. This module provides methods to multiply 2-dimensional to 1-dimensional and accepts any type of data ranging from integer to double. It has several overloaded methods to handle all types of data and all types of arguments.

Fully qualified class name: *dsss*.util.Correlator

Correlator: This module provides all the methods needed for correlating two signals or methods for auto correlation. This also provides correlating function with respect to sampling rate which gives more accurate correlation. It supports all kinds of arguments.

Fully qualified class name: *dsss*.util.Integrator

Integrator: This module integrates the input signal and gives an integrated output. The parameters accepted by the methods are a 2- dimensional array or a 1- dimensional array with code length. If this is a 2-dimensional array it integrates all the columns in a row and divides with the row length. The result is a 1-dimensional double array with the length of row length. If the provided parameter is a 1- dimensional array with the code length then it integrates a set of code length bits and gives the output. The resultant array is total array length/ code length.

Fully qualified class name*:* *dsss*.util.Quantizer

Quantizer: This module provides all the functionality for quantizing the signal. The provided quantizer is hard decision quantizer where it decides zero or one based on whether the input value is greater than zero or less than zero. The level can be dynamically set by setting the parameters.

Fully qualified class name: *dsss*.util.Shifter

Shifter: This class provides the shifting of the signal right side or left. This module is mainly used in generating m sequences where the shifting is very necessary. This is used

in the correlator module to shift the signal. The methods in the module can shift right or left with definite shift values.

Package: *dsss*.interleaver:

This package is used to generate an interleaved array which can be used to interleave the generated data sequence and code words. This package provides two classes one for convolutional interleaving and other for random interleaving.

Fully qualified class name: *dsss.interleaver.Convolution*

Convolution Interleaver: It interleaves the given array into a new bit array by interleaving convolutionally. The algorithm followed to interleave the bits is by convolution. The parameter to interleave the bit array is the number of rows used to interleave. This class is useful when we use the coding technique as convolutional coding.

Fully qualified class name: *dsss.interleaver.Random*

Random Interleaver: This interleaves the given array in a random process. The *java.util.Random* class is used to generate the random sequence.

Package : *dsss*.generator

This package has all the classes to produce various types of sequences like Gold sequence generator, PN Sequence Generator, Gaussian Noise Generator, m-sequence generator and orthogonal code generator. The entire generator is capable of generating both 2-level or ordinary codes and 3-level or generalized codes.

Fully qualified class name: *dsss.generator.GoldSequence*

GoldSequence: This class produces gold sequence for a respective length. This has several methods to produce the gold sequence for different levels.

Fully qualified class name: *dsss.generator.Orthogonal*

Orthogonal: This generator generates both 3-level and 2-level orthogonal signal by using exhaustive search algorithms. This can even produce semi-orthogonal signals with given maximum and average deviation. The algorithm of exhaustive search is explained more in Chapter 4.

Fully qualified class name: *dsss.generator.3level_MSequence*

3level_MSequence: This class is responsible to produce m-sequences for 3-level. The algorithm used for producing m-sequences for 3-level is explained in [4]. This class accepts LFSR tap connections and LFSR offset and generates the sequence. The tap connections must be given appropriate to generate m-sequence. The tap connections can be obtained from [4] for generating m-sequences.

Fully qualified class name: *dsss.generator.2level_MSequence*

2level_MSequence: This class is same as the above class except that the produced sequence is 2-level instead of 3-level.

Fully qualified class name: *dsss.generator.NoiseGenerator*

Noise Generator: This class produces various types of noise and interference. The possible noises produced by the class are Gaussian and canonical. The output produced by Gaussian generator has a P_E (PDF) of Gaussian distribution and for canonical it is canonical distribution. The generator is also able to produce multi-path interference. The function can produce multi-path interference for a specific signal and specific SNR. The

algorithm used to produce the interference is explained in Figure 17: Flow Chart for generation of multi-path interference.

Package: *dsss*.application.scenario

This package has all the implementation of all the scenarios used in the thesis. This package gives the sample implementation of the scenario and usage of the library function. There are implementations for Gaussian, multi-path and multiple-access scenario. This class provides sample implementation and can be extended to create more complex situations.

Fully qualified class name: *dsss.application.scenario.level2.Gaussian*

This class has a runnable program for simulating 2-level Gaussian scenario as shown in Figure 9 (without normalization and code generator uses 2-level code instead of 3-level). The program accepts data length and code length for the simulation and produces BER output for SNR ranging from 0 to 10. The output is written to the file and to the console.

Fully qualified class name: *dsss.application.scenario.level2.multi.Multipath*

This class has a runnable program for simulating 2-level multi-path scenario as shown in Figure 16 and the interference is generated by using multi-path interference flow chart described in Figure 17. The code generator in Figure 17 generates a code word with 2-levels (+1 and −1 amplitude). The program accepts data length, code length, code sequence type and SNR range and generates BER for SNR. The output is written to the external file in the file system and console. The code sequence type is random or m-

sequence. For random code modulation the codes generated are randomly generated from the java.util.Random class, for m-sequence the program asks for the number of shift registers to be used to generate m-sequence to use in the simulation.

Fully qualified class name: *dsss.application.scenario.level2.MultipleAccess*

This class has a runnable program for simulating 2-level multiple-access scenario as shown in Figure 30. The program accepts data length, code length, number of users and code type. The code type can be of random, orthogonal and semi-orthogonal codes. The random code selection assigns a random code for each user in the system which does not guarantee minimum cross correlation between users. The orthogonal code selection assigns orthogonal code for each user which is generated using "Hadamard" algorithm [8] which guarantees zero cross correlation between codes. The semi-orthogonal selection assigns semi-orthogonal codes (generally gold sequences or sequences from file) to each user.

Fully qualified class name: *dsss.application.scenario.level3.Gaussian*

This class has a runnable program for simulating 3-level Gaussian scenario as shown in Figure 9 (with normalization and code generator uses 3-level code instead of 2-level). The program accepts data length and code length for the simulation and produces BER output for SNR ranging from 0 to 10. The output is written to the file and to the console.

Fully qualified class name: *dsss.application.scenario.level3.multipath.Multipath*

This class has a runnable program for simulating 3-level multi-path scenario as shown in Figure 16 and the interference is generated by using multi-path interference flow chart described in Figure 17. The code generator in Figure 17 generates a code

word with 3-levels (+1, 0 and −1 amplitude). The program accepts data length, code length, code sequence type, SNR range and generates BER for SNR. If the type of code sequence is m-sequence the program asks for the number of shift registers used to generate 3-level m-sequences.

Fully qualified class name: *dsss.application.scenario.level3.MultipleAccess*

This class has a runnable program for simulating 3-level multiple-access scenario as shown in Figure 30. The program accepts data length, code length, number of users and code type. The code type can be of random, orthogonal and semi-orthogonal. The random code selection assigns a random code for each user in the system which does not guarantee minimum cross correlation between users. The orthogonal code selection assigns orthogonal code for each user which is accepted from the file. The semi-orthogonal selection assigns semi-orthogonal codes (Generally sequences from file) to each user.

CHAPTER III

MULTI-PATH FADING CHANNELS

3.1 Introduction

Multi-path is a simple term referring to the multiple paths through which a radio wave can propagate between transmitter and receiver. This propagation includes ground wave, reflection from earth, ionosphere reflection, reflection from the buildings and so on. As a result the receiver sees the superposition of the same signal with different phases, different attenuation and delays. This can result in constructive or destructive interference, amplifying or attenuating the signal power at the receiver. This effect can cause errors and affect the quality of the system [8]. Figure 12 illustrates the multi-path scenario and effect of multi-path scenario.

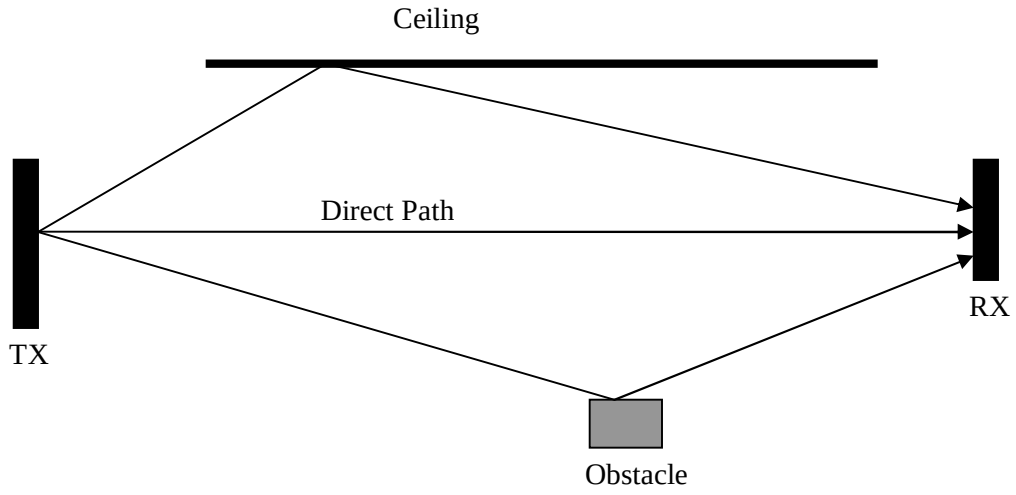

Figure 12: Multiple Paths

In the above diagram there are 3 paths from TX to RX. The path labeled "Direct Path" indicates the direct path and the other paths are reflected from the ceiling and an obstacle. Due to this the receiver RX has several signals with different phase angles, different amplitudes and different delays. The scenario can be further explained clearly in Figure 13.

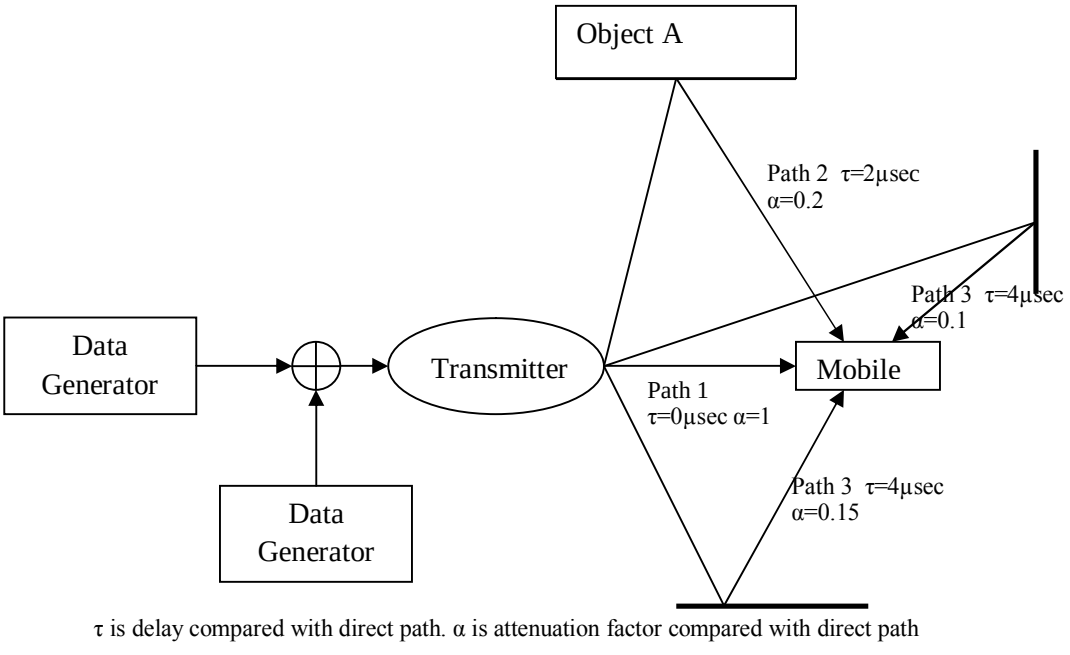

Figure 13: Multi-path fading

In the Figure 13 path 1 is a direct path. As direct path is the shortest distance between transmitter and receiver and the delay with respect to shortest path is $\tau = 0$ μsec (excluding propagation delay) and attenuation factor is equal to 1 as direct path has minimum attenuation. The remaining 3 paths are reflected paths from several terrains or buildings. Each individual reflected path arrives at its own amplitude and delay. The path amplitude of each reflected path depends on relative propagation distance and reflective or refractive properties of the terrain or building. In Figure 13 path 2 has a delay of 2μsec and attenuation factor of 0.2. The attenuation factor depends upon propagation distance and properties of the Object A. This causes the received signal to have more reflected paths and the energy due to the reflected paths may be much higher than the direct path. For wide band (digital) signal the main effect of multi-path fading is dispersion and inter-symbol interference. This interference can cause many errors and seriously degrade the performance of the communication system. The channel model of multi-path is shown in Figure 14.

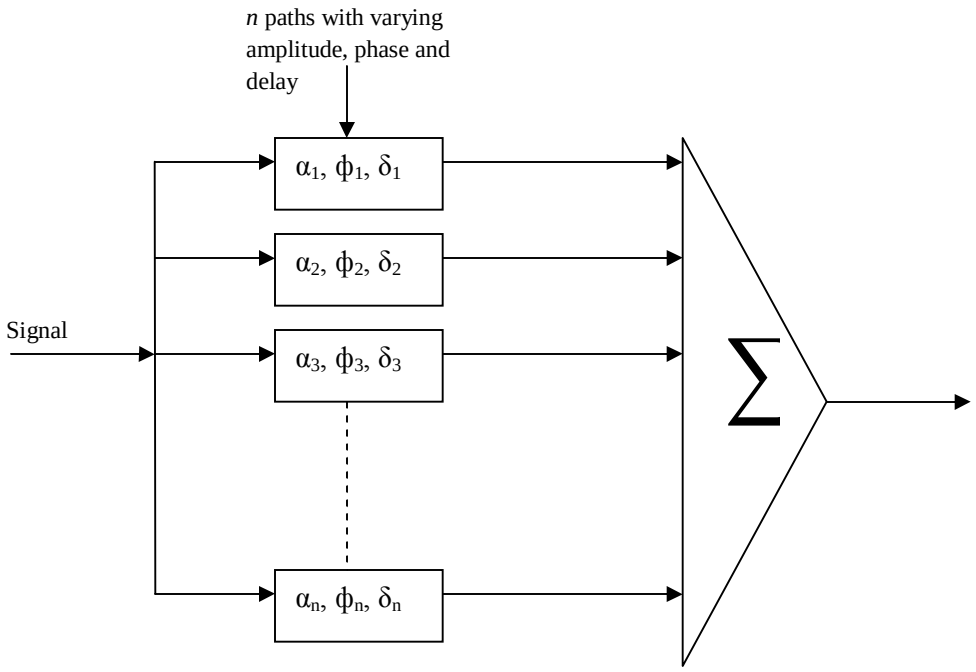

Figure 14: Propagation model for multi-path signals

In Figure 14 there are *n* path delays where *n* is total number of paths from transmitter to receiver. Here, α is attenuation factor which is dependent on the reflective and refractive index of the reflecting object. As the reflecting objects can be anything like ground, ionosphere, terrain or a building the attenuation and delay is random. So all the signals add as random vectors with the amplitude of each term appearing to be Rayleigh-distributed and the delay distributed uniformly.

3.2 Delay Spread

Delay spread has various definitions. One of the definitions is that the maximum delay of the first direct signal (line of sight) to last arrived signal at the receiver is called a

delay spread. The other definition of the delay spread is the standard deviation or root mean square (RMS) value of all multi-path delays for a given signal.

Calculation of delays for delay modules are determined by the delay spread used for the system. If the simulation is done in Macro Cellular conditions then according to [12] the delay is about 20μsec.

For $20\,m\sec$ *path length difference between direct path and reflected path is*

$$3\times10^{8}\times20\times10^{-6}=6000 \text{ meters}.$$

If the chip frequency is $1MHz$ *each chip is* $\frac{3\times10^{8}}{10^{6}}=300 \text{ meters}.$

For a 6000 *meters the no of chips to be delayed are* 20

From the above calculation we should have a delay block which delays about 20 chips. Each delay block shown in Figure 16 have different delay but the RMS of the delay blocks is equal to 20 chips. For having more precise delay the signal is sampled further and delayed. The delay block shown in the Figure 16 cannot delay if the delayed signal is less than 1μsec. To attain the delay for less than 1μsec the chip is sampled with an even larger rate and the samples are delayed.

Several scenarios can be considered like urban, sub-urban and rural; each has their own standard deviation, which is shown in

Figure 15 15. According to [12] the delay spread for various locations is shown in Figure 15.

Environment	Delay Spread (μsec)
Macro Cellular (Rural Flat)	0.5
Macro Cellular (Urban)	5
Macro Cellular (Hilly)	20
Micro Cellular (Factory, Mall)	0.3
Micro Cellular (Indoor, Offices)	0.1

Figure 15: Multi-path delays

3.3 Block Diagram for Multi-Path Scenario Simulation

The block diagram for the scenario explained in Figure 13 is shown in Figure 16. The block diagram is same as in Gaussian scenario explained in Figure 9 except the interference generated. All the modules used in this simulation are same modules used in Gaussian scenario. The interference block contains a set of delay-attenuation blocks which delays the signal by τ msec and attenuates the signal by α factor. The delay and attenuation values are calculated by given RMS and SNR value. The calculations are explained below.

Calculation of delay signal (τ): The delay module accepts number of paths and RMS delay spread value to be delayed and returns an integer array with all delays. The delays are returned such that RMS (or average) value of integer values is equal to input RMS.

Example: $RMS\,Value = 30\mu\sec$

$Number\ Of\ Paths = 4$

Select first 3 *delays randomly with average to be* $30\mu\sec$ *say*

$\tau_1 = 20\mu\sec, \tau_2 = 40\mu\sec$ *and* $\tau_3 = 20\mu\sec.$

Now calculate τ_4 *such that RMS value is equal to* $30\mu\sec$

$$30 = \sqrt{\frac{\tau_1^{\,2} + \tau_2^{\,2} + \tau_3^{\,2} + \tau_4^{\,2}}{4}}$$

$$900 \times 4 = 20^2 + 40^2 + 20^2 + \tau_4^{\,2}$$

$$\tau_4 = 47\mu\sec$$

Calculation of attenuation factor (α): The attenuation factor is calculated from given SIR (Signal to Interference Ratio). The module accepts SIR, number of paths and returns a double array with all attenuation factors.

Example: $SIR = 5db$

$Number\ Of\ Paths = 4$

Calculate α_1, α_2, α_3 *and* α_4

$$10\log_{10}\left(\frac{S_{pow}}{I_{pow}}\right) = 5\,db \quad where\ S_{pow} = Signal\ Energy\ and\ I_{pow} = Inteference\ Energy$$

$$\frac{S_{pow}}{I_{pow}} = 10^{0.5}$$

$$\frac{S_{pow}}{I_{pow}} = 3.1$$

$$\frac{S_{amp}}{I_{amp}} = \sqrt{3.1} = 1.77 \quad where\ S_{amp}\ and\ I_{amp}\ are\ Signal\ amplitude\ and\ Interference\ amplitude$$

$$I_{amp} = \frac{S_{amp}}{1.77}$$

$$I_{amp} = 0.56 * S_{amp}$$

Now the attenuation factors for individual path are calculated such that their total is equal to 0.56

Sample attenuation factors for four paths are given below

$\alpha_1 = 0.13$, $\alpha_2 = 0.17$, $\alpha_3 = 0.16$ *and* $\alpha_4 = 0.20$

By the resultant values from above two modules the interference is calculated by using the algorithm explained in the next section and added to the signal. The resultant signal is given to the demodulator (or receiver) to demodulate the signal. The demodulator is same as explained in Gaussian scenario case.

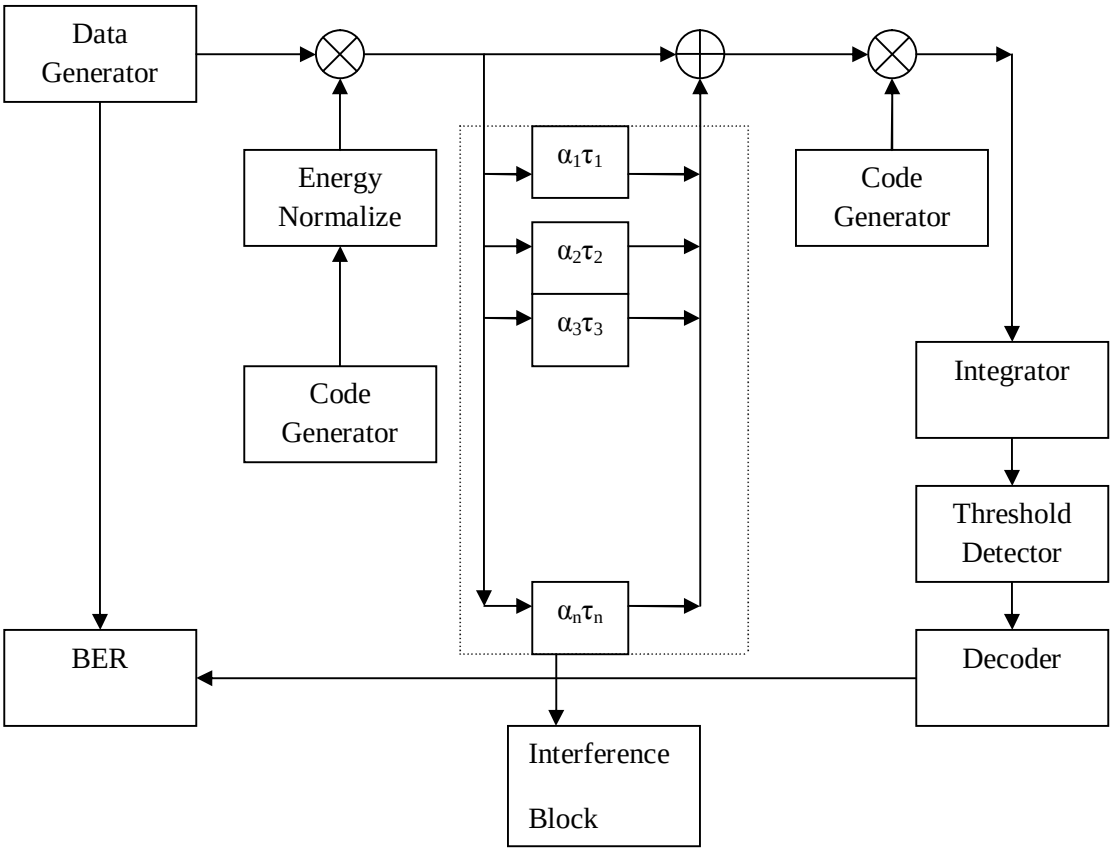

Figure 16: Block diagram of multi-path scenario

The demodulation is same as explained in the Gaussian scenario. The simulation is done for various delay spreads and various modems like random modem and m-sequence modem.

3.4 Algorithm and Flow Chart for Generating Multi-path Interference

1) Set the average deviation, set the Random Class with that average deviation.
2) Set the SIR, Initialize interference array.
3) Calculate number of paths and attenuation factor for each path to produce the required SIR.
4) Set path number = 1.
5) Generate deviation from the above initialized class.
6) Calculate number of samples to be delayed.
7) Delay input signal by that number of samples and multiply by attenuation factor.
8) Add the signal to interference array.
9) Increase the path number to +1.
10) If the path number < number of paths go to step 3 else continue.
11) Feed the resultant interference to the simulation.
12) End.

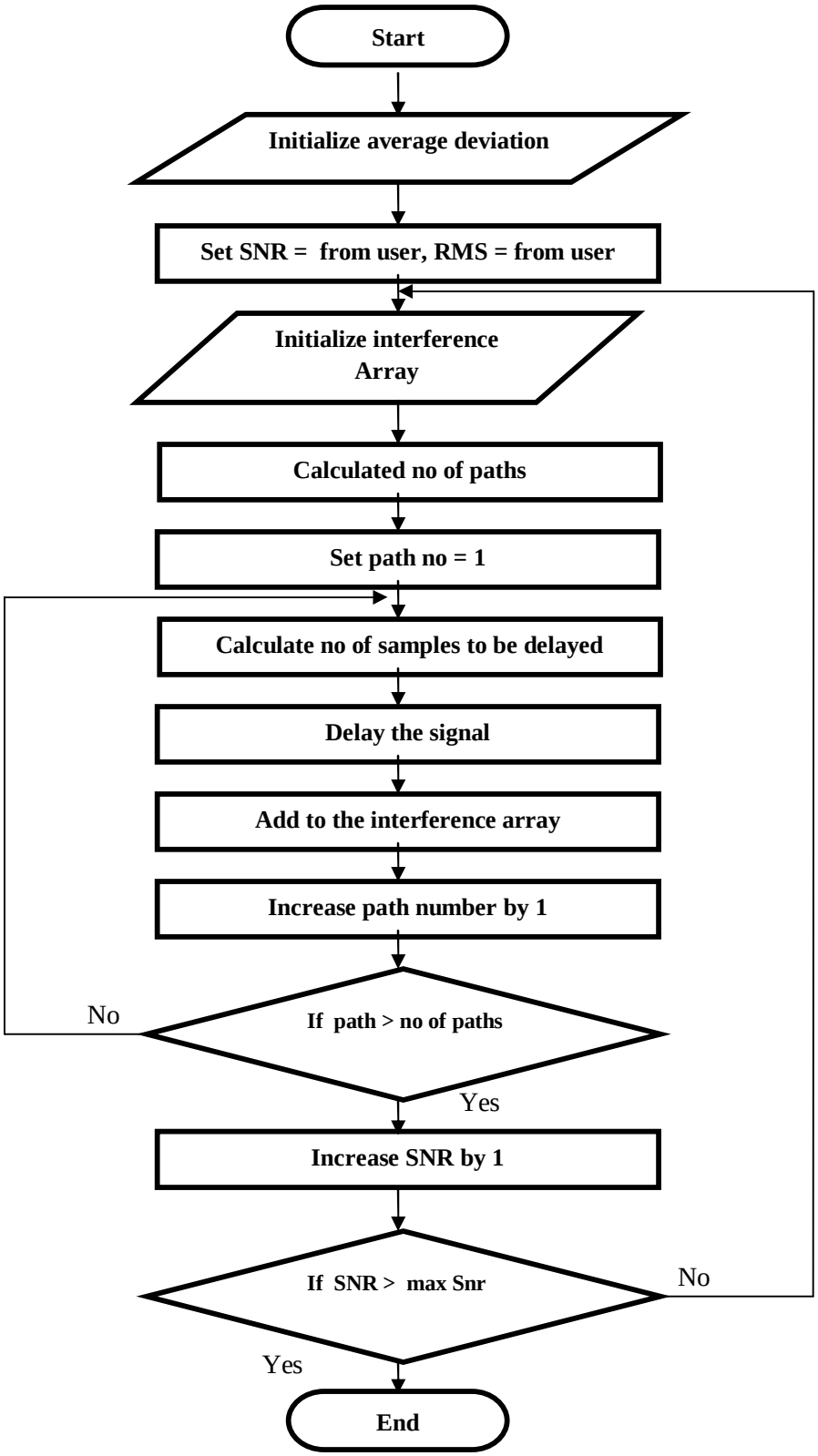

Figure 17: Flow Chart for generation of multi-path interference

In this thesis we considered three cases:

- Modulating base signal with Random Code as signature sequence.
- Modulating base signal with m-sequence as signature sequence (1 MHz as m-sequence frequency).
- Modulating base signal with m-sequence as signature sequence (10 MHz as m-sequence frequency).

3.5 Modulating Base Signal with Random Code as Signature Sequence

In the random code the PN sequence or signature sequence used in the system was a random code. The code is produced by giving a unique seed for the random class in Java which ensures a pseudo random code. So the randomness and auto-correlation property are unknown. The delay used in the delay blocks has a standard delay of 20 chips. All the delays produced by the system are added and feed to the receiver.

3.5.1 Simulation Results

The simulation result shows that the performance of generalized spread spectrum is not better than the performance of ordinary spread spectrum. In the graph shown in Figure 18 the performance of generalized spread spectrum is worse than the performance of its counterpart, ordinary spread spectrum.

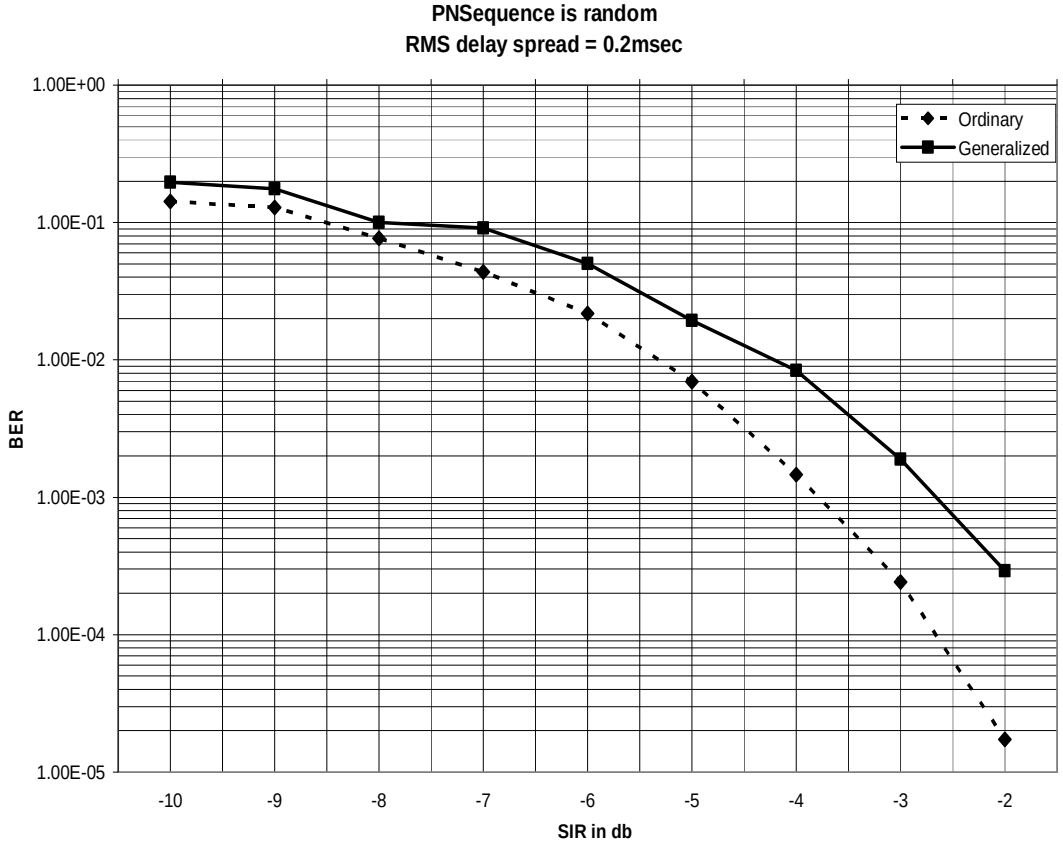

Figure 18: Simulation results of random code scenario

We can observe from Figure 18 that the ordinary direct sequence spread spectrum performs well beyond generalized direct sequence spread spectrum. The performance is even better at high SIR for ordinary direct sequence spread spectrum compared to generalized direct sequence spread spectrum.

3.6 Random Modem with M-sequences as Spreading Sequences

In a practical communication model (like IS-95 [1]) the signature sequence used in direct sequence spread spectrum systems are m-sequences because of their auto

correlation properties. So for this thesis we have simulated the communication system with the standard m-sequences. The pseudo random sequences or signature sequences used must possess some characteristics which are stated below.

- The number of runs of 0's and 1's is equal. We want equal number of two 0's and 1's, a length of three 0's and 1's and four 0's and 1's, etc. This property is required for a perfectly random sequence.
- There are equal number of runs of 0's and 1's. This ensures that the sequence is balanced.
- The periodic autocorrelation function (ACF) is nearly two valued with peaks at 0 shifts and is zero elsewhere. This allows us to modulate the signal effectively and using the ACF peak to demodulate quickly. To create this type of sequences m-sequences are widely used in digital communication.

3.6.1 Introduction to M-sequences

Maximum length sequences also known as m-sequences are one type of pseudorandom sequences. These sequences are the basis for generating pseudo random sequences in digital communication systems like DS-CDMA. These sequences have all the characteristics needed for DS-CDMA. For this thesis we have used ordinary m-sequences (2-level) for simulating ordinary multi-path scenario and generalized m-sequences (3-level) for simulating generalized spread spectrum multi-path scenario. In the following subsections we will discuss about both m-sequences, their generations, and proceed with the simulation results.

3.6.2 Ordinary or 2-level M-sequences

The sequences are ordinary m-sequences and are used in current communication systems. They have only 2 levels $\{-1, +1\}$. These sequences can be created using a shift-register with feedback-taps. By using a Single shift-register, *maximal length sequences* can be created and called often by their shorter name of *m-sequence*, where m stands for maximal.

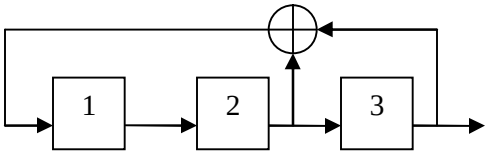

Figure 19: An LFSR for producing 2-level m-sequences

In Figure 19 a LFSR is shown which generates m-sequence of period 7, using taps after shift register 2 and 3. M–sequences are created using **linear feedback registers (LFSR)**. Figure 19 shows a three register LFSR with tap connection arrangement. The tap connections are based on primitive polynomials in the order of the number of registers and unless the polynomial is irreducible, the sequence will not be an m-sequence and will not have the desired properties. Each configuration of L registers produces one sequence of length $2^L - 1$. If taps are changed, a new sequence is produced of the same length. There are only a limited number of m-sequences of a particular size. The periodic auto correlation has only two values and is given by [8]:

$$A(K) = \begin{cases} 1.0 & k = N \\ \dfrac{1}{-N} & k \neq N \end{cases} \quad \textit{Where N is period of } m-\textit{sequence}$$

The graph shown in the Figure 20 is auto correlation of m-sequence over an entire sequence. As the m-sequence has only two values $\left\{1, \frac{1}{-N}\right\}$, to be more precise between the transitions from 1 to $\frac{1}{-N}$ the sequence is further sampled by 10 times by a sampler. So the x-axis in the graph shows the cyclic shift of samples but not the m-sequence chips. Due to its two-valued auto correlation function the multi-path effect is very low as the delay signal has negligible energy when compared to original given the delay is more than one chip.

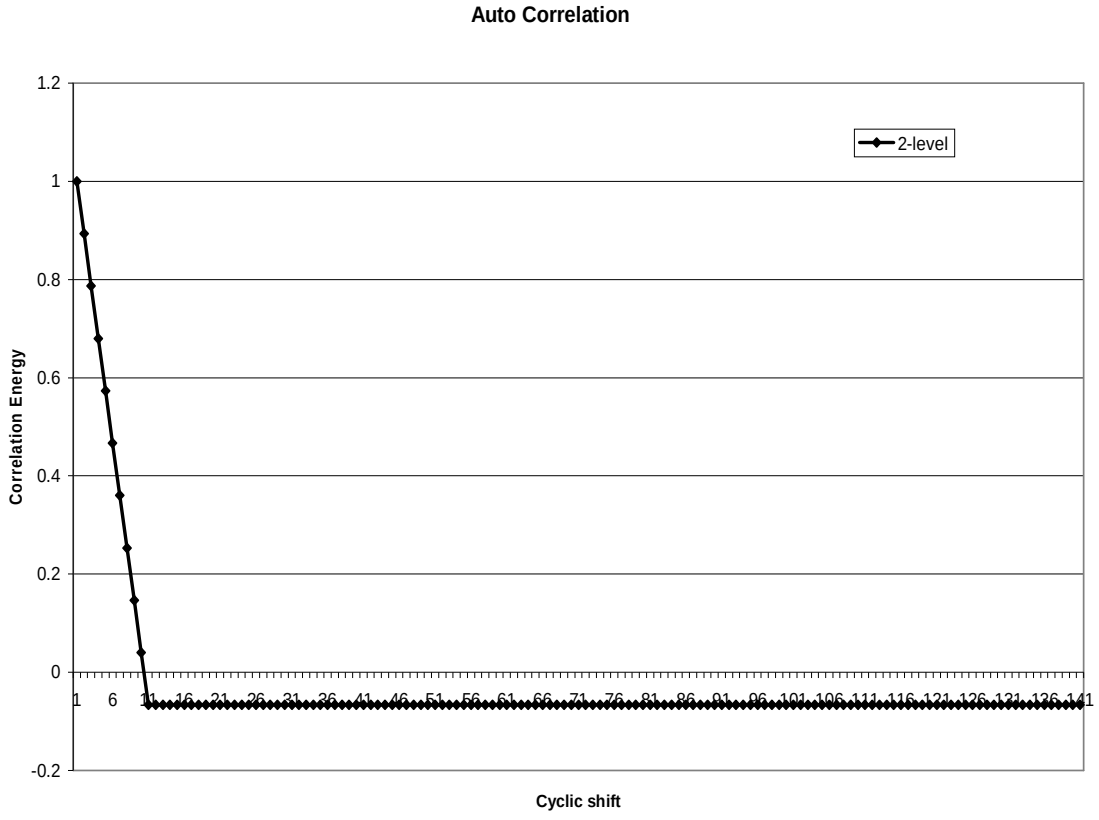

Figure 20: Auto-correlation for 2-level m-sequence signal

3.6.3 Generalized or 3-level M-sequences

As we are dealing with generalized spread spectrum we must have the equivalent m-sequences. In [4] there is proof and generation of m-sequences for generalized direct sequence spread spectrum. These sequences will be produced by the shift feedback register with the tap functions based on polynomial expression over GF(3), i.e., Galois field over 3. So the tap coefficient has more than two constants. To produce the 3-level m-sequences the coefficients may be 0, 1 or 2 [4].

Consider the polynomial expression $f(x) = x^4 + 2x^3 + 2x^2 + x + 2$

The view of linear feedback shift register would be as shown in Figure 21.

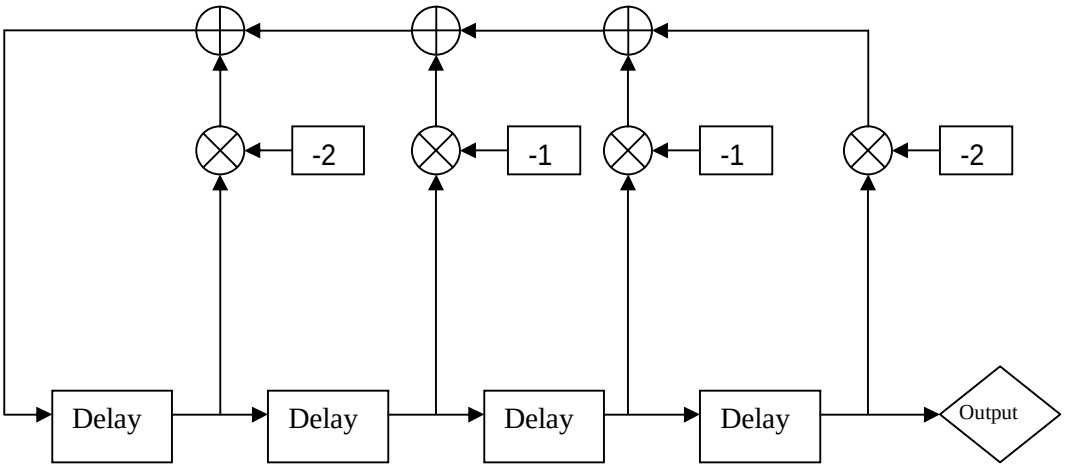

Figure 21: LFSR used to generate 3-level m-sequence

But the addition and multiplication for 3-level sequences is different from 2-level sequences. The rules for addition and multiplication is shown in Figure 22 and Figure 23.

+	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

Figure 22: GF (3) Addition

X	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

Figure 23: GF (3) Multiplication

By using above circuit and above calculation we can generate the m-sequences for 3-level. The auto-correlation for the sequence produced by the m-sequences is three-valued instead of two-valued for ordinary m-sequences.

For the shift of 26 the auto correlation (when it matches itself) = 0.675

For the shift 13 the auto correlation = -0.675

For all the other values the auto correlation = 0

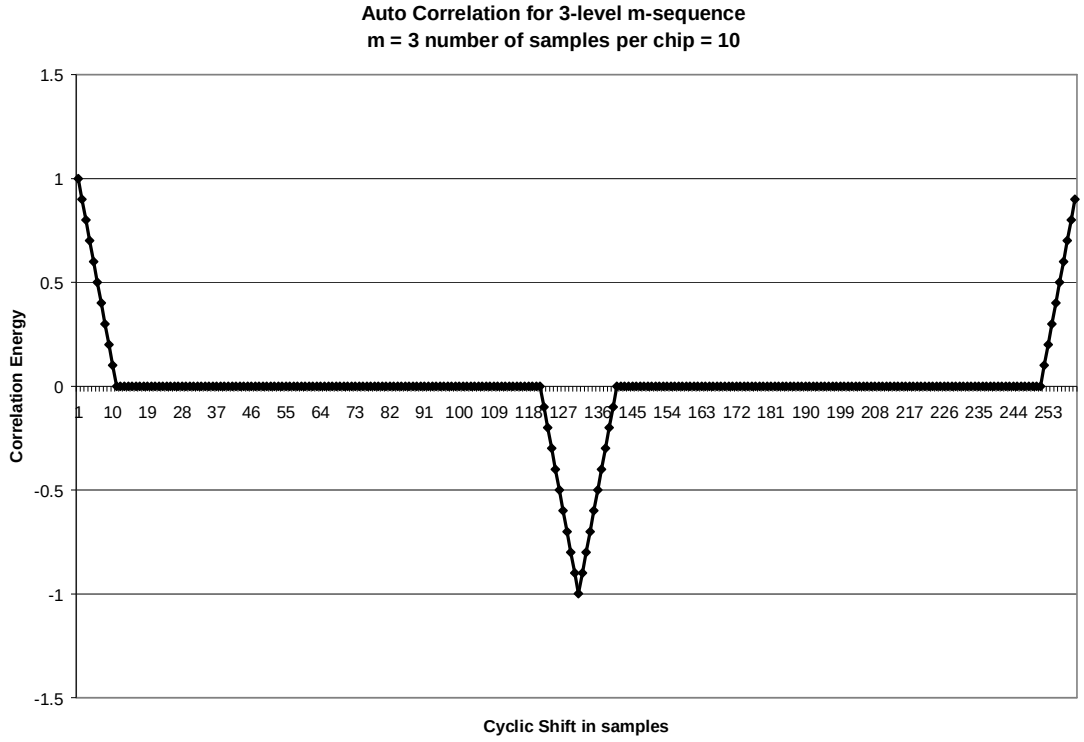

Figure 24: Auto-Correlation of 3-level m-sequences

The graph shown in Figure 24 is the 3-level auto correlation graph, which illustrates the difference between ordinary m-sequences and generalized m-sequences. From the graph shown in Figure 25 if the delay is greater than one chip length then the performance of generalized spread spectrum would be better than the ordinary spread spectrum as the auto correlation for generalized m-sequence is 0 compared to ordinary m-sequence -0.01.

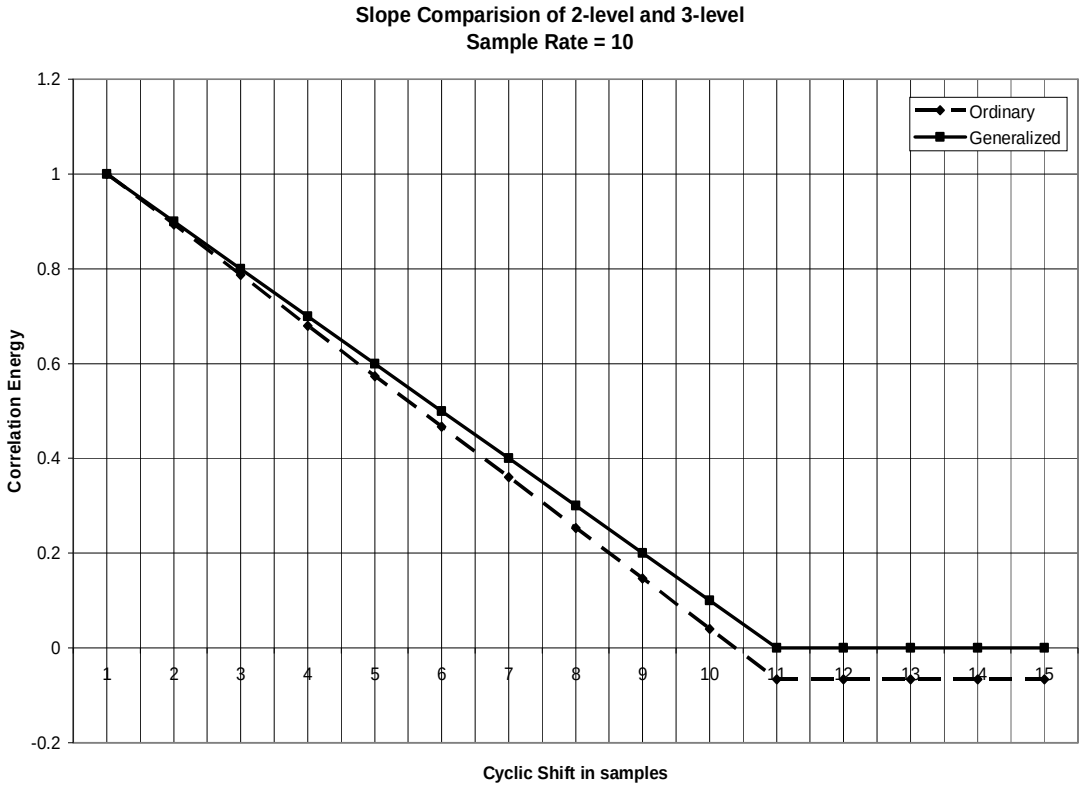

Figure 25: Comparison between 2-level and 3-level auto correlation

3.6.4 Simulation Results for Low Frequency

For practical communication model the chip frequency used is 1MHz and the RMS delay spread would be 0.2μsec. For the delay blocks to have average delay of one chip the signal is even sampled to 10 samples per chip and the samples are delayed with the average delay spread of 10 samples. The m-sequence for 3-level is generated by five tap Linear Feed Back Registers (LFSR) (which can produce up to 3^5 = 273 length m-sequence) and for 2-level eight tap LFSR's are used (which can generate up to 2^8 = 256

length m-sequence). As both of them have approximately equal length of m-sequence we have selected the 5 and 8 shift registers for 3-level and 2-level respectively. This makes the simulation more accurate. In Figure 26 we can see that performance for ordinary is better than generalized direct sequence spread spectrum.

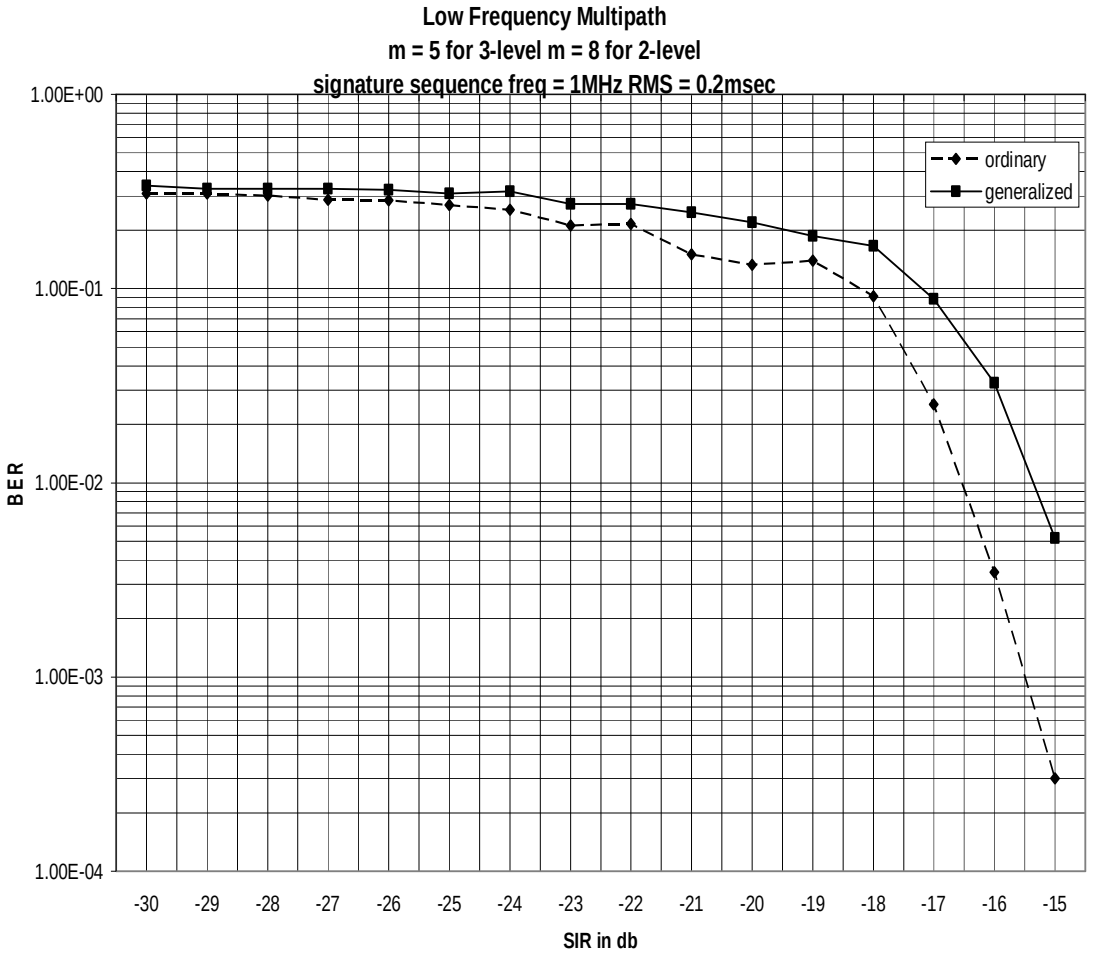

Figure 26: Simulation results for ordinary and generalized spread spectrum for low frequency communication model in multi-path fading interference

3.6.5 Simulation Results with High PN Code Sequence Frequency

In this simulation we considered the code rate which would be much more than 1 MHz i.e. about 10 MHz. This makes the delay spread across 10 chips. The graph shown in Figure 27 is by considering high code frequency. The m-sequence for 3-level and 2-level is same length as used for above simulation.

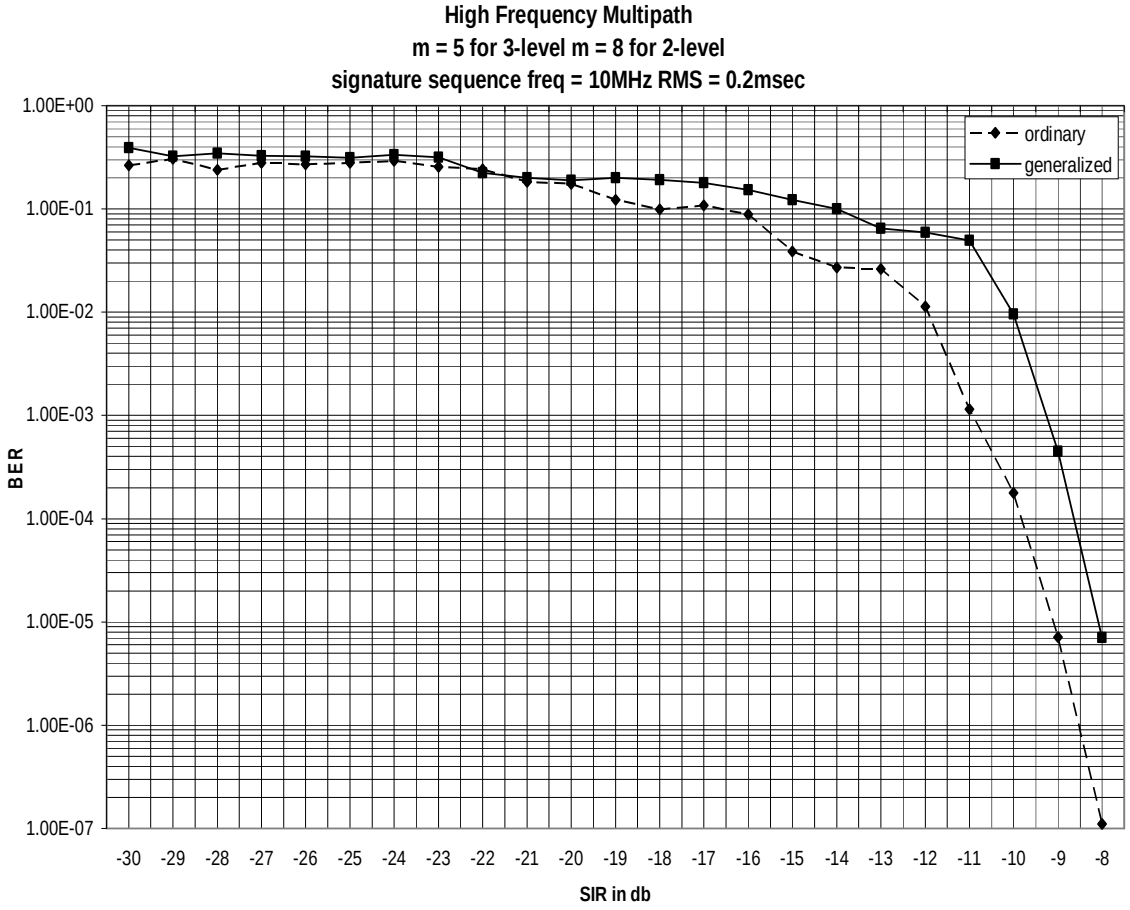

Figure 27: Simulation results of ordinary and generalized spread spectrum by using high frequency m-sequence in multi-path fading

From the graph shown in Figure 27 it is evident that at low SIR the performance is somewhat same but as the SIR increases the performance for ordinary direct sequence spread spectrum is better than generalized direct sequence spread sequence.

3.7 Conclusion:

We have created a simulation environment for a channel corrupted by multi-path fading and simulated both ordinary and generalized spread spectrum in various scenarios. We have simulated by having random pseudo noise as signature sequences and found that ordinary spread spectrum is performing better than generalized spread spectrum. We have also simulated with m-sequences as signature sequences with different frequencies, different delay spreads and compared both of the results and found that ordinary direct sequence spread spectrum performs better than generalized direct sequence spread spectrum.

CHAPTER IV

MULTIPLE-ACCESS

4.1 Introduction

This section introduces some of the basic concepts like multipe-access, orthogonal codes, semi-orthogonal codes and CDMA.

4.1.1 Multiple-Access

Multiple-access is nothing but capability of accessing the same channel at the same time by multiple users. There are several techniques to allow access to the channel like FDMA, TDMA and CDMA.

4.1.2 Orthogonal and Semi-Orthogonal Codes

Orthogonal Codes are codes which have the zero cross correlations, that is the dot product of vector representation of the codes have a 90 degree angle. In general the cross correlation and dot product are mathematically related. Any code string can be

represented as a vector. A code of 1111 can be represented by the vector of (1,1,1,1) where each bit can be taken as a separate component on the vector [1]. Calculation of the angle between two codes are shown below.

Consider two codes 1111 and 1010. Their vector representations are (1,1,1,1) and (1,0,1,0).

Signal representation of Code A is (1,1,1,1) and Code B is $(1,-1,1,-1)$.

$$|A| = 1^2 + 1^2 + 1^2 + 1^2 = 4$$

$$|B| = 1^2 + (-1)^2 + 1^2 + (-1)^2 = 4$$

$$A.B = (1\times 1) + (1\times -1) + (1\times 1) + (1\times -1) = 0 = cross\,correlation$$

$$Angle\,\theta = \cos^{-1}\frac{A.B}{|A|.|B|}$$

$$\theta = \cos^{-1}\frac{0}{4\times 4} = 90^0$$

So the above two vectors are orthogonal and their cross correlation is 0. All the orthogonal signals have the cross correlation value as zero. Semi orthogonal signals have small cross correlations that depend on the angle between the two signals.

4.1.3 Code Division Multiple Access

Code division multiple-access techniques allow many users to simultaneously access a given frequency allocation. User separation at the receiver is possible because each user spreads the modulated waveform over a wide bandwidth using unique spreading codes. Direct-sequence CDMA (DS-CDMA) spreads the signal directly by multiplying the data waveform with a user-unique high bandwidth pseudo-noise binary

sequence. The resulting signal is then mixed up to a carrier frequency and transmitted. The receiver mixes down to base band and then re-multiplies with the binary {± 1} pseudo-noise sequence. This effectively removes the pseudo-noise signal and what remains (of the desired signal) is just the transmitted data waveform. After removing the pseudo-noise signal, a filter with bandwidth proportional to the data rate is applied to the signal. Because other users do not use completely orthogonal spreading codes, there is some residual interference [1].

DS-CDMA spreads the transmitted signal by directly multiplying the signal with a unique high bandwidth pseudo noise. The resultant signal is then mixed with the carrier signal and then transmitted to the receiver. At the receiver side the received signal is multiplied by the same pseudo noise and converted to base band signal. But as all users transmit at the same time there is possibility of having multiple-access interference. To eliminate it all the other users must use perfectly orthogonal pseudo noise. Perfectly orthogonal pseudo noise is impractical; however, there are several techniques to have orthogonal signals. For perfectly orthogonal codes, the well-known sequence generator is Walsh codes. For semi-orthogonal codes, a Gold Sequence generator is used.

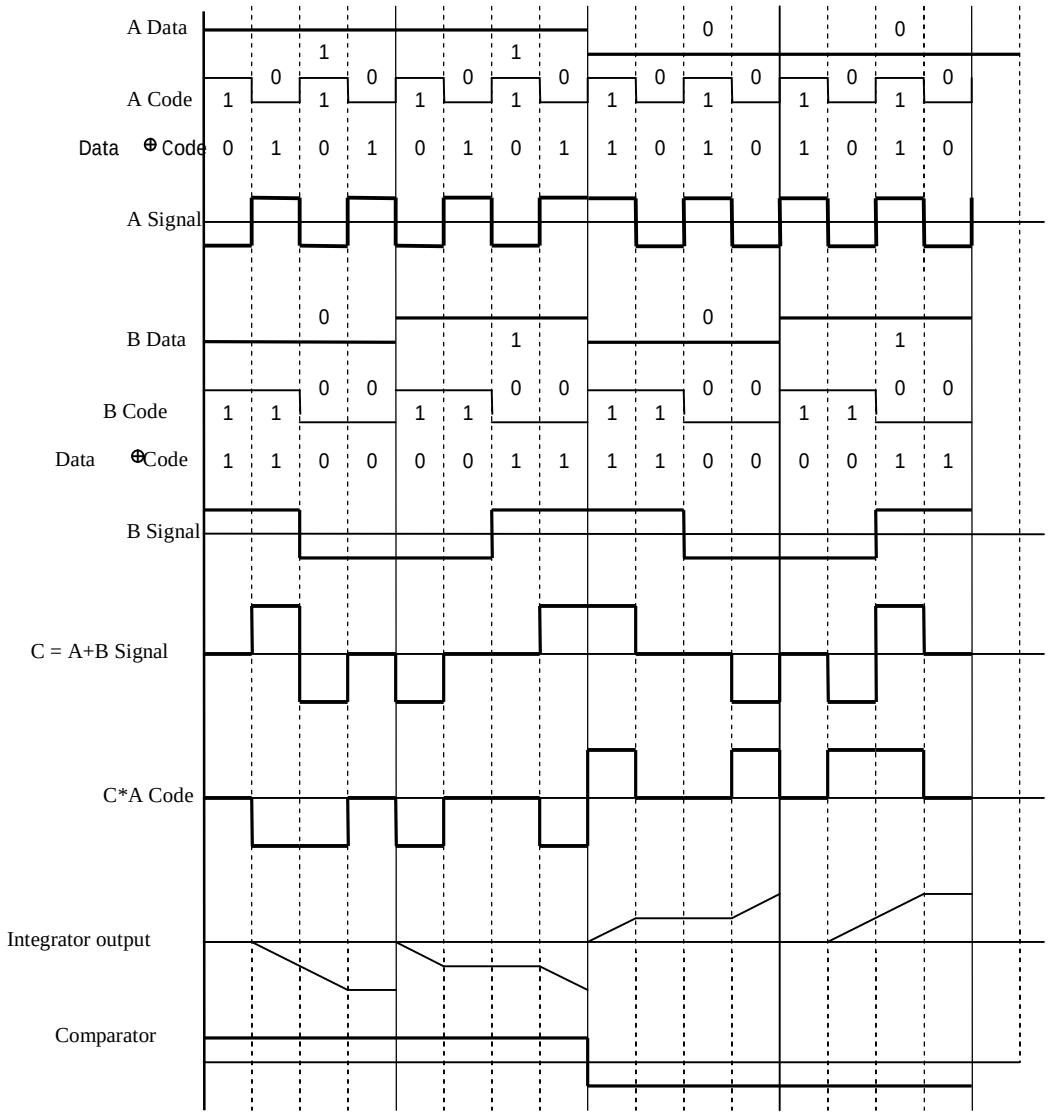

Figure 28: Signal Representation of CDMA system

In the Figure 28, signal representation of a CDMA system shows where two users share same channel but modulated and demodulated with perfectly orthogonal codes. In Figure 28, user 'A' uses (1010) as the code word and user 'B' uses (1100) as the code word which is perfectly orthogonal to the code used by user 'A'. The two users have multiplied their data to the code word and converted into signal form where 0 is changed to (-1) amplitude and 1 is changed to (+1) amplitude. The resultant signals of 'A' and

'B' are added to form signal C as shown in Figure 28. At the receiving side (mobile unit) the signal C is again multiplied by Code A and integrated over a period of code length which results in signal D shown in Figure 28. The result is given to a comparator or threshold detector where it decides the bit as 1 if the integrated value is greater than 0 and as 0 if the integrated value is less than 0. As one can see, the resultant output is exactly same the data transmitted for user 'A'. Even the integrated value is the same value without the signal B added because of code B being perfectly orthogonal to code A.

In Figure 29 user 'A' and user 'B' use code words which are not perfectly orthogonal to each other. User 'A' uses code word (1010) and user 'B' uses code word (1110) which are not orthogonal and has a dot product of value of (+1) amplitude. As the codes are not orthogonal the integration value (signal D) is not same as the integration value (signal D) in Figure 28. This is due to multiple access interference from signal B. As the dot product of the cross correlation between the codes increases the interference increases effecting overall SIR ratio and BER. Smaller the deviation from 90 degrees between codes, smaller interference is and larger the deviation from 90 degrees between codes, larger the multiple access interference is. This forms the basis for this chapter.

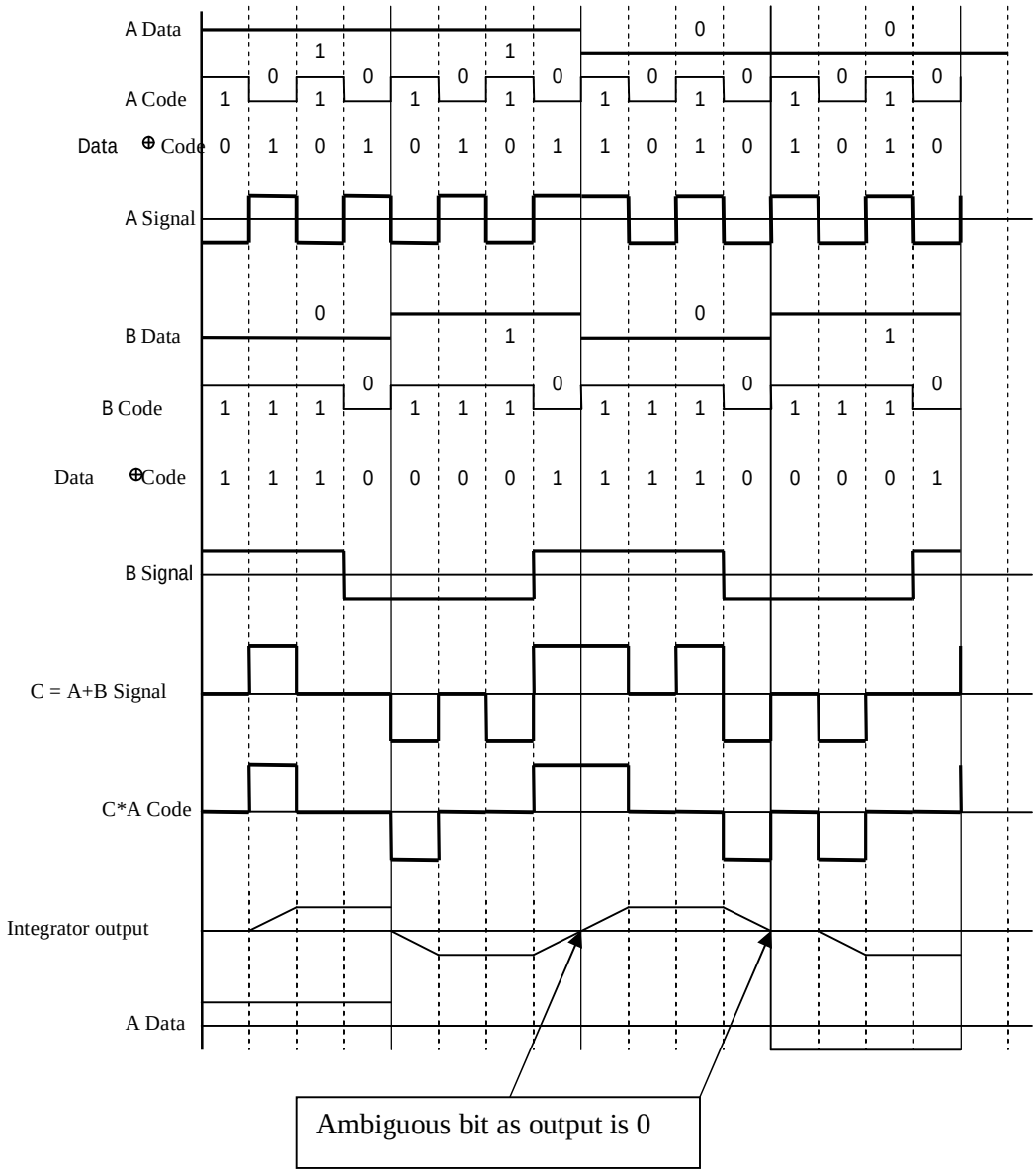

Figure 29: Signal representation of CDMA system with semi orthogonal codes used

4.2 Multiple-Access with Random Codes

In Figure 30, a random generator with different and unique seeds generates the codes. This makes the cross correlation between the pseudo generators a minimum. At the receiver side the same seed used for generating first code is used to generate the code.

This allows the same sequence to be generated at the receiving end as at the transmitting end.

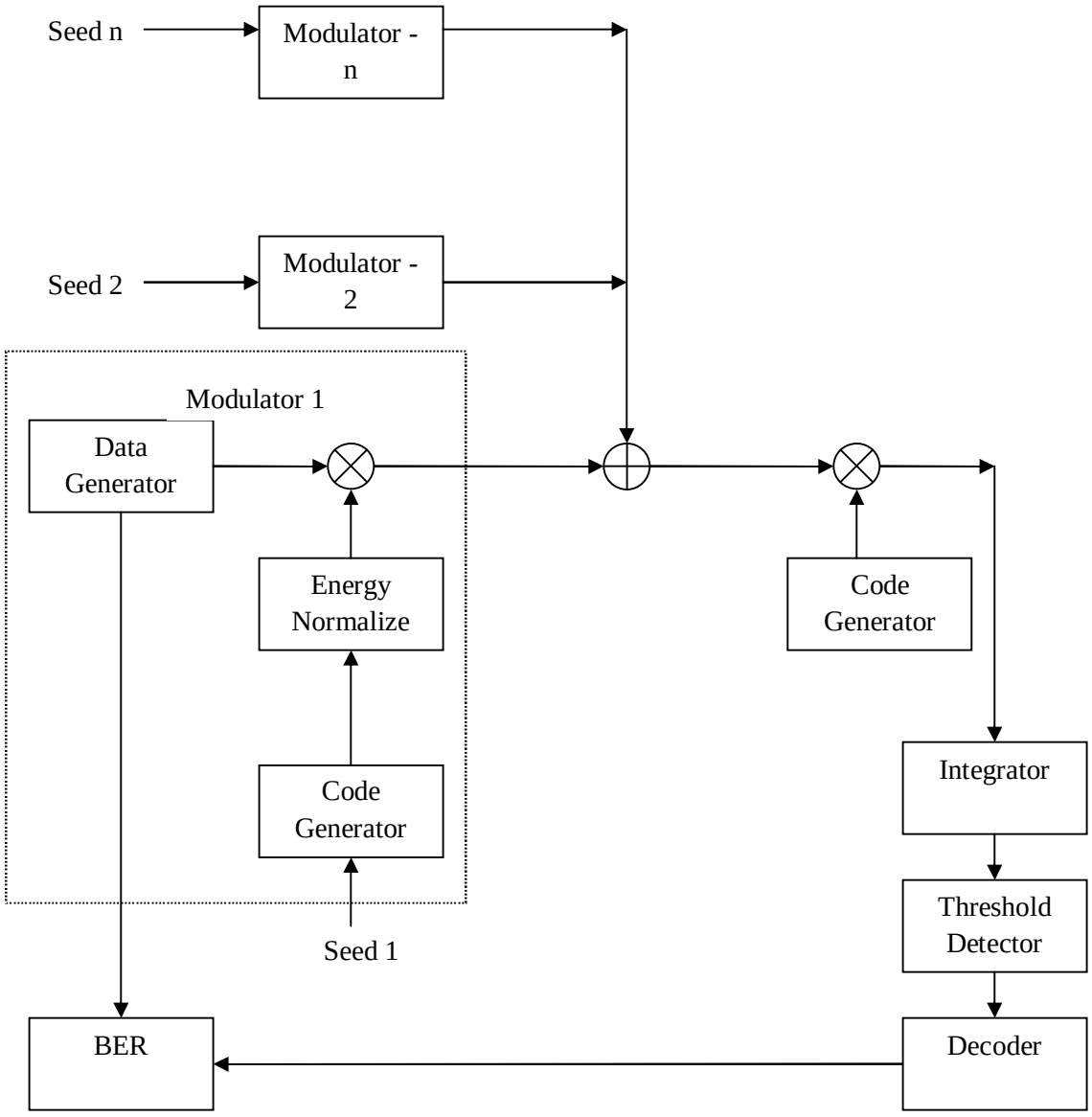

Figure 30: Block Diagram of multiple-access scenario for random codes

For the simulation the results are taken by increasing the number of user from 8 to 40 for a code length of 20. This makes sure of testing the simulation from smallest multi-

access interface to highest multi-access interface. The results for ordinary and generalized spread spectrum are shown in Figure 31. From the graph shown in Figure 31 it is evident that both the ordinary and generalized spread spectrum perform same as the addition of random codes from different users makes the interference behave like Gaussian noise.

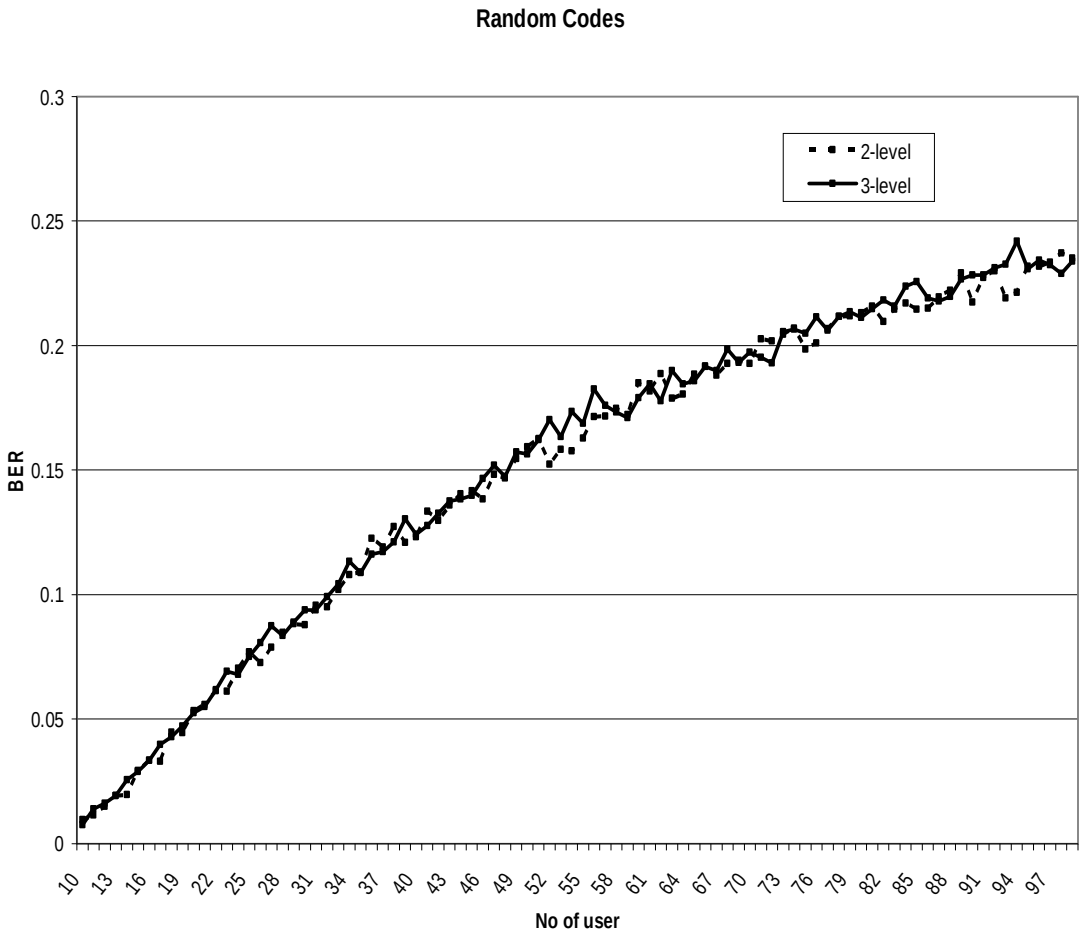

Figure 31: Simulation results for ordinary and generalized spread spectrum in multiple-access with random codes assigned to users

4.3 Multiple-Access with Orthogonal Signals

For simulating with perfectly orthogonal signals the first step is to generate perfect orthogonal codes for ordinary and generalized spread spectrum. For ordinary or 2-level spread spectrum, Walsh Codes are used to get perfect orthogonal signals.

4.3.1 Orthogonal Signal for Ordinary or 2-level Sequence

Walsh codes are created out of Hadamard matrices and Transform. Hadamard is the matrix type from which Walsh created these codes. Walsh codes have just one outstanding quality: In a family of Walsh codes, all codes are orthogonal to each other.

The matrix used to generate Walsh codes are shown below.

$$H_{N+1} = \begin{bmatrix} H_N & H_N \\ H_N & -H_N \end{bmatrix}$$

Where initial H_1 matrix is

$$H_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

and all the higher matrices are derived from the previous matrices.

4.3.2 Orthogonal Signals for Generalized or 3-level Sequences

As the **Hadamard** matrices are defined by binary sequences there is no way of producing perfectly orthogonal signals for generalized spread spectrum. For this purpose we have used an exhaustive search algorithm which was used for generating semi orthogonal signals (described in 4.6 Exhaustive Search).

4.3.3 Block Diagram

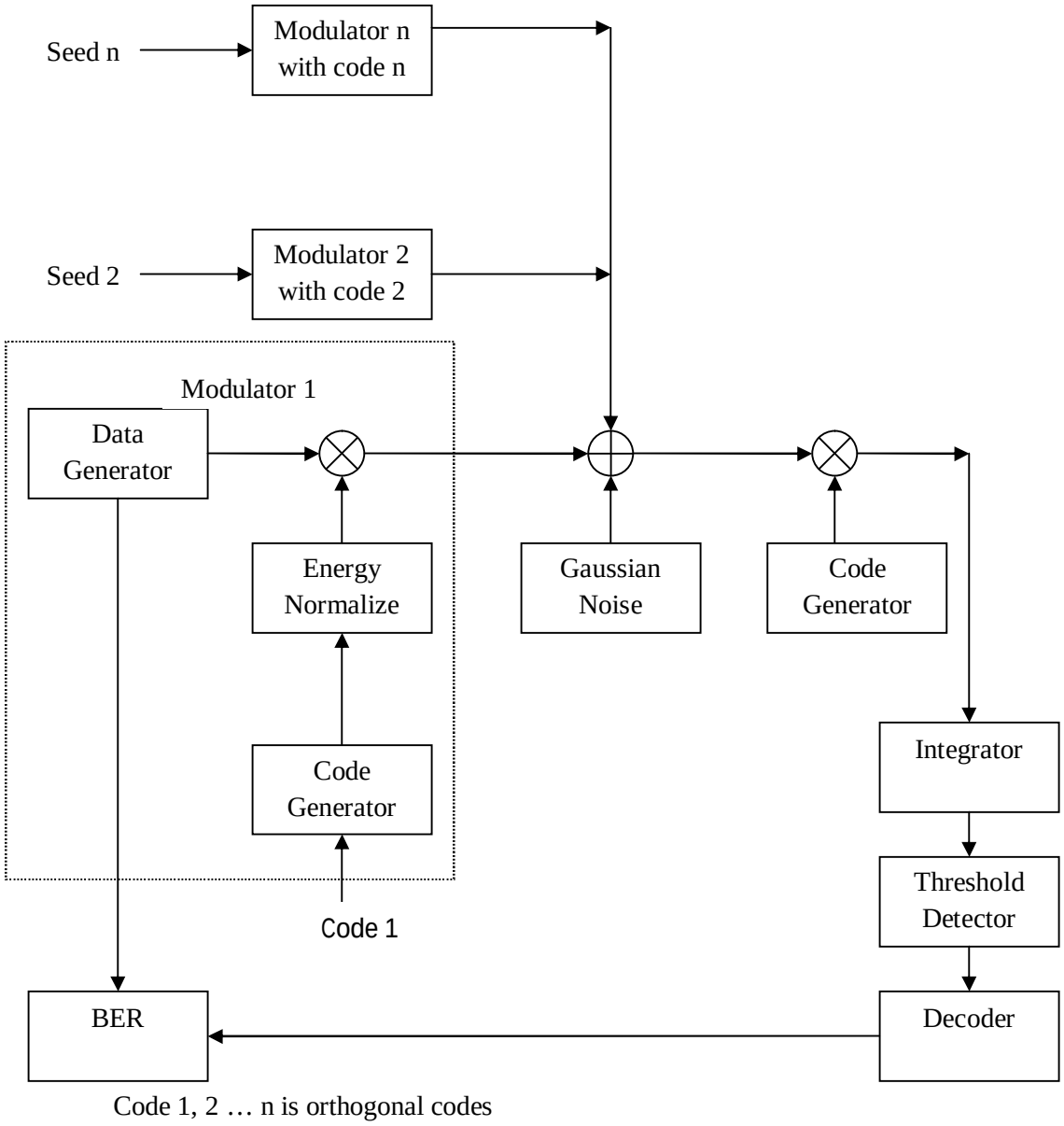

Figure 32: Block Diagram of multiple-access scenario with orthogonal codes

In Figure 32, the codes used for every code generator is a predefined code which form a perfectly orthogonal set. The output shown in Figure 33 suggest that there is no

difference in 2-level and 3-level because for perfectly orthogonal signals only Gaussian noise is present and no multiple access interface. This makes the graph to be same for both 2- level and 3–level systems. But if we consider for each code length, 3-level has more number of points where orthogonally can occur. For 2-level, perfectly orthogonal codes occur only at code length of 2, 4, 8, 16, 32 but for 3-level, the points occur at 2, 4, 6, 8, 12, 16. But for same code length, the maximum number of perfectly orthogonal codes will be same for both 2-level and 3-level.

4.3.4 Simulation Results

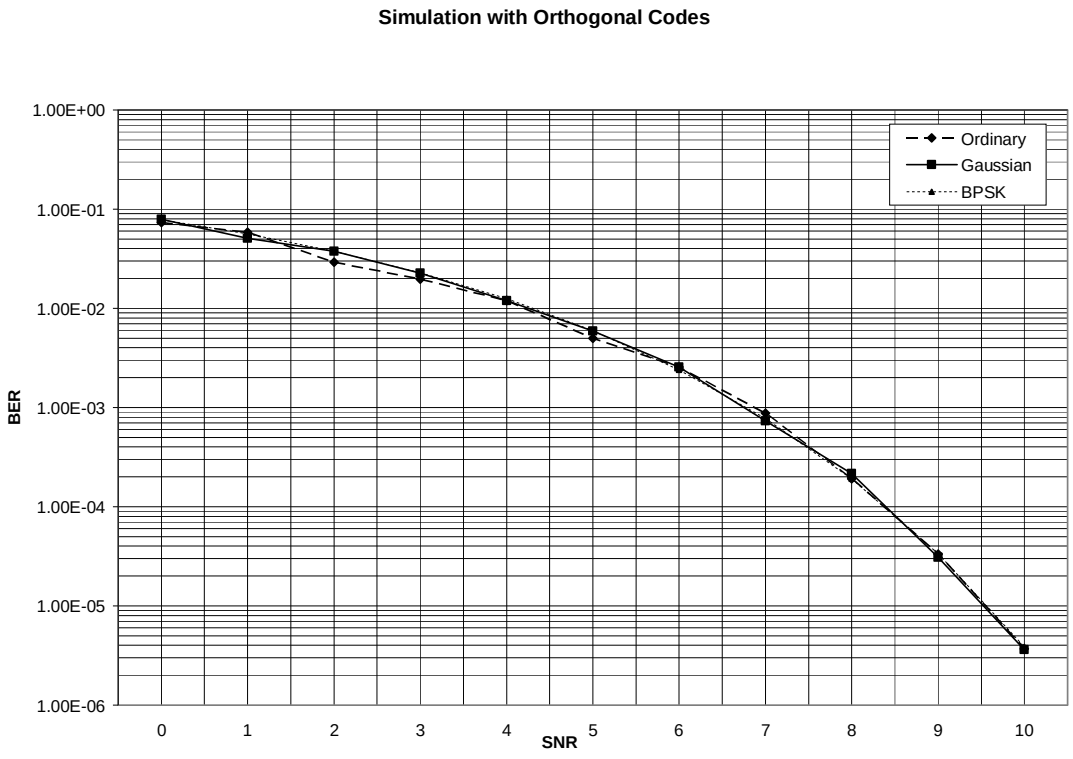

Figure 33: Simulation results for ordinary and generalized spread spectrum in multiple-access scenario with orthogonal codes assigned to users

From Figure 33 we can observe that for orthogonal signals the 2-level and 3-level have same performance as the theoretical graph of BPSK. This is because for perfect orthogonal signals the only noise present in the system is thermal noise. There would be no multiple access interference present.

4.4 Simulation with Semi Orthogonal Signals

This section talks briefly about the simulation of the multiple access using semi orthogonal signals. The main advantage of 3-level over 2-level are its maximum number of code words for a particular length. This can lead to more semi-orthogonal signals found for 3-level than 2-level. This is briefly explained in the next chapter. For this section we only inspect the block diagram, simulation results and conclusion. The block diagram for the simulation is shown in Figure 34.

4.4.1 Block Diagram

In Figure 34 all the modules are same as in the orthogonal multiple access case except now the codes used in the code generators are semi orthogonal. For generating semi orthogonal signals there are several algorithms for 2-level sequences, such as Gold sequences, etc. However, there is no pre defined algorithms for 3-level sequences. We approached in several ways to generate effective 3-level semi orthogonal codes. The following section provides all the approaches we followed. The two main approaches are:

- Applying existent algorithms to 3-level.
- Exhaustive search with different parameters.

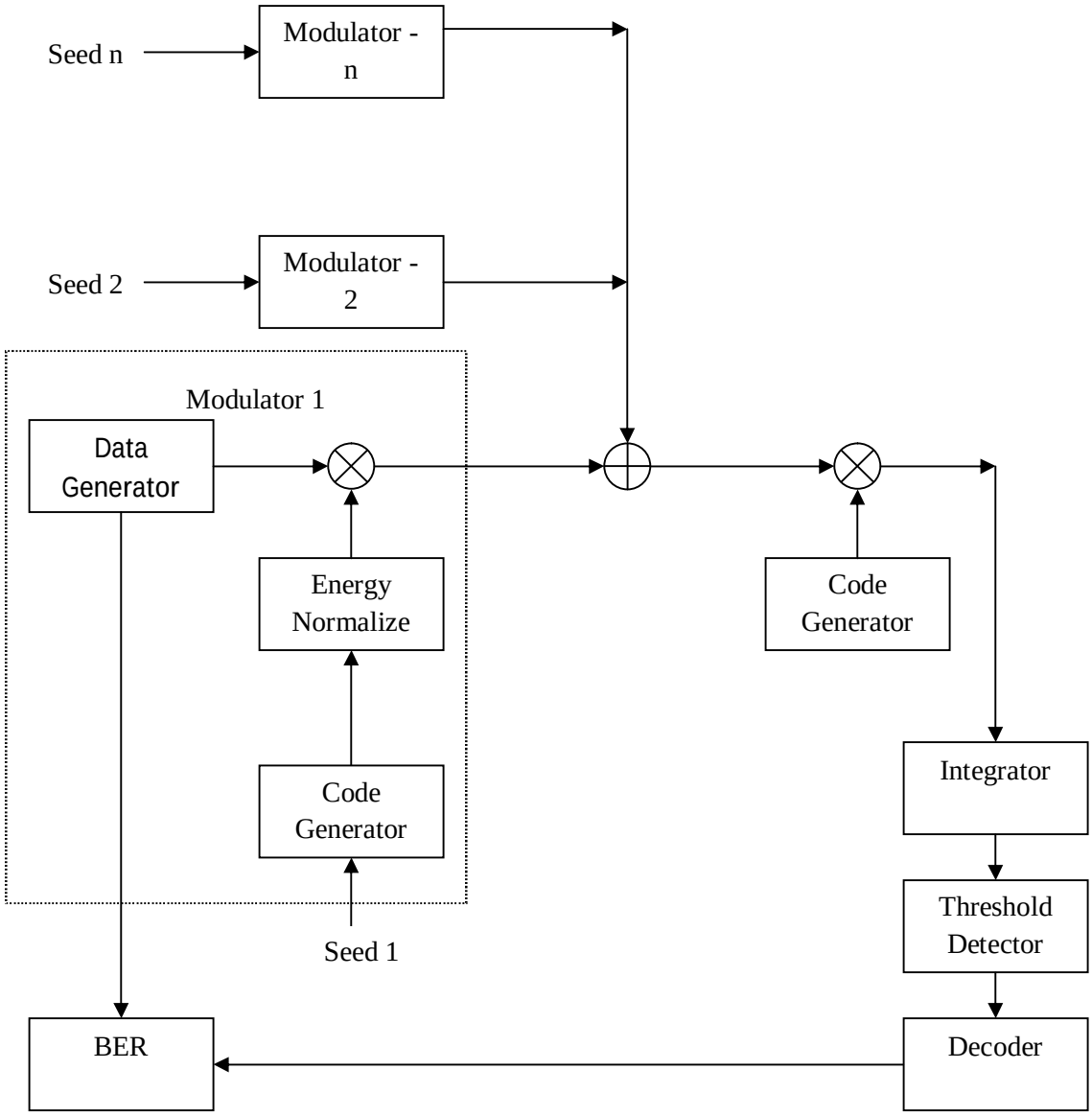

Figure 34: Block diagram of multiple-access scenario with semi-orthogonal signals

4.5 Applying Existence Algorithm

In this section we will discuss about generating semi orthogonal signal for 3-level through Gold sequence.

4.5.1 Gold Sequences for 2-level

Gold Sequences were proposed by Gold in 1967 [8]. Gold codes are constructed by XOR-ing two preferred m-sequences of the same length with each other. These codes are used in asynchronous CDMA systems. Gold sequences are an important class of sequences that allow construction of long sequences with three valued Auto Correlation Functions. Gold sequences are constructed from pairs of preferred m-sequences by modulo-2 addition of two maximal sequences of the same length.

Gold sequences are useful in non-orthogonal CDMA. Gold sequences have only three cross-correlation peaks, which tend to get less important as the length of the code increases. They also have a single auto-correlation peak at zero, just like ordinary PN sequences. The use of Gold sequences permits the transmission to be asynchronous. The receiver can synchronize using the auto-correlation property of the Gold sequence.

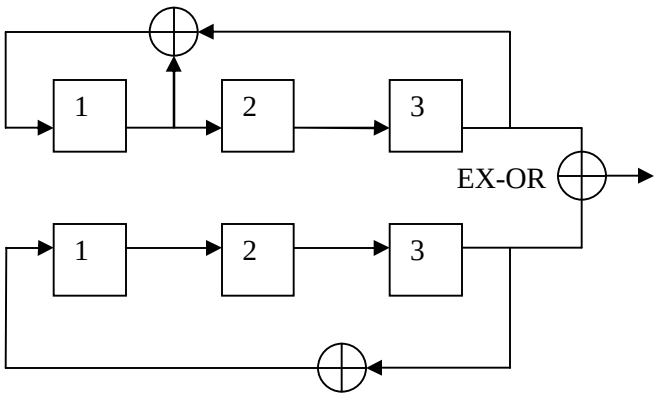

Figure 35: Generation of Gold sequences with preferred pair of m-sequences

There are several algorithms present to generate 2-level orthogonal and semi-orthogonal sequences. Gold sequences are used widely to generate semi orthogonal

signals. Some pairs of m-sequences with the same degree can be used to generate Gold codes by linearly combining two m-sequences with different offsets in Galois field. All pairs of m-sequences do not yield Gold codes and those which yield Gold codes are called *preferred pairs*. Gold codes have three-valued autocorrelation and cross-correlation functions with values $\{-1, -t, t-2\}$.

$$t = 2^{(m+1)/2} + 1 \quad m \text{ is odd}$$
$$t = 2^{(m+2)/2} + 1 \quad m \text{ is even}$$

The generation of Gold codes is very simple. Using two preferred m-sequence generators of degree *r*, with a fixed non-zero seed in the first generator, 2^r Gold codes are obtained by changing the seed of the second generator from 0 to 2^r-1. Another Gold sequence can be obtained by setting all zeros to the first generator, which is the second m-sequence itself. In total, (2^r+1) Gold codes are available. For generating Gold sequences we applied the generated m-sequence from above algorithm to Gold sequence generator which is used to generate 2-level sequences. As shown in Figure 36 the angle between different pair of signal is constant and had a value of 91^0 where the deviation is 1^0 from perfectly orthogonal signals.

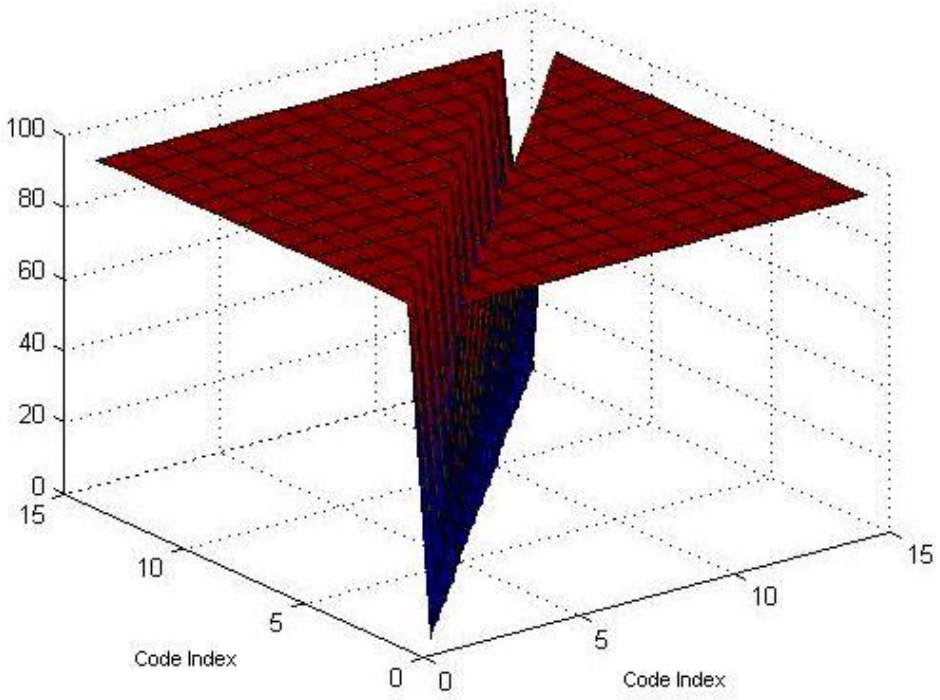

Figure 36: Angle of all 2-level gold sequences

4.5.2 Gold Sequences Generation for 3-level:

The above concept is applied for generation of 3-level. For generating Gold sequences there are two requirements:

- Find m-sequences for 3-level which have same characteristics as 2-level.
- Find preferred m-sequences.

For the first step we have used m-sequences by using algorithm provided by [4]. But the characteristics of 3-level is not exactly same as 2-level. It does not have constant auto correlation instead it has negative auto correlation at the center of the series.

For the second step there are no preferred m-sequences for 3-level which has three valued auto correlation. The best possible preferred m-sequences have 7-valued auto correlation. These two above reasons make the generation of 3-level Gold sequences difficult using this method.

The graph in Figure 37 shows the angle between all the signals formed by implementing Gold sequences algorithm on generalized m-sequences with seven valued auto correlation.

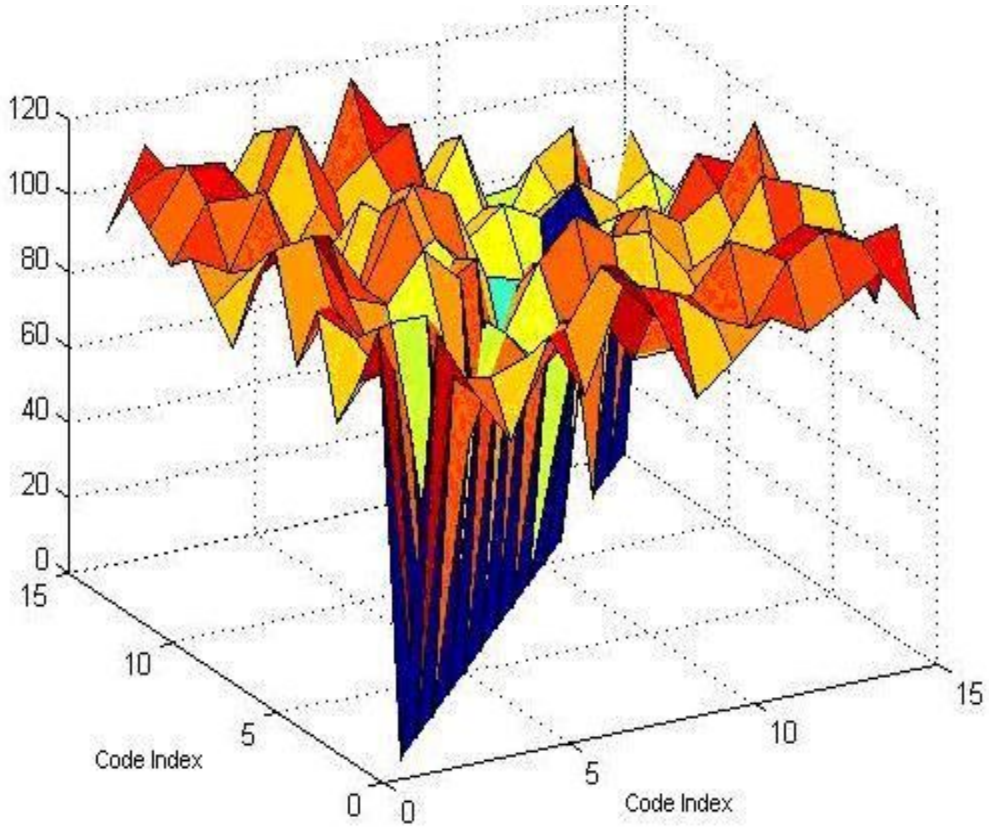

Figure 37: Angle of all 3-level gold sequences

In the Figure 37, it is evident that the angles for different pair of signals have large deviation from 90^0 (perfectly orthogonal) and ranges from 80^0 to 105^0. So the deviation

is not evenly distributed across all the sequences. Some of the produced sequences have very high deviation from 90^0, which is intolerable for multiple-access. This is mainly because of the characteristics of m–sequences produced by 3-level.

From the graphs shown in Figure 20 and Figure 24 we can observe that the 2-level m-sequence takes two values $\{1\ ,\ -1/N\ \}$ but for 3-level the auto correlation takes three values $\{1,\ 0\ ,\ -1\}$. This makes the sequences produced by the gold sequence generator to have high cross correlation. And the other reason for not having good cross correlation signals is the nature of preferred sequences. For 2-level there are several sequences which had auto correlation with only 3 values shown above. But no two 3-level sequences have such behavior. The sequences used in the generator of 3-level sequences do not have 3-valued auto correlation property. Instead it has 5-valued auto correlation which made the output sequences to have more cross correlation.

4.6 Exhaustive Search

As there is no mathematical model to generate semi-orthogonal signals for 3-level we tried to generate semi-orthogonal signals by computer based exhaustive search. Exhaustive search is nothing but searching for orthogonal signals by comparing all the possible codes that can be formed for a particular length. The computer program is designed such that it compares all possible sets of semi-orthogonal signals to have best set of semi-orthogonal signals, which has low average auto correlation and low maximum correlation between any two codes.

4.6.1 Algorithm for Exhaustive Search

Below is the algorithm for generating semi-orthogonal signals by computer-based or exhaustive search.

1) Accept code length..
2) Start Timer.
3) Initialize average deviation to 1.
4) Set the start sequence to all 1's.
5) Add the start sequence to orthogonal vector array.
6) Set the new sequence to be start sequence.
7) Measure average deviation and max deviation between the present sequence and all sequences present in orthogonal vector.
8) If average deviation is less than user deviation and max deviation is less than user specified deviation, add the vector to orthogonal vector array.
9) Increase the sequence by 1.
10) Go to step 6 until the sequence returns to start sequence.
11) Store the formed sequences in a local array and increase deviation by 1.
12) Go to step 2 until the deviation reaches user specified max deviation.
13) End Timer and measure the amount of time for a given code length.
14) End the process.

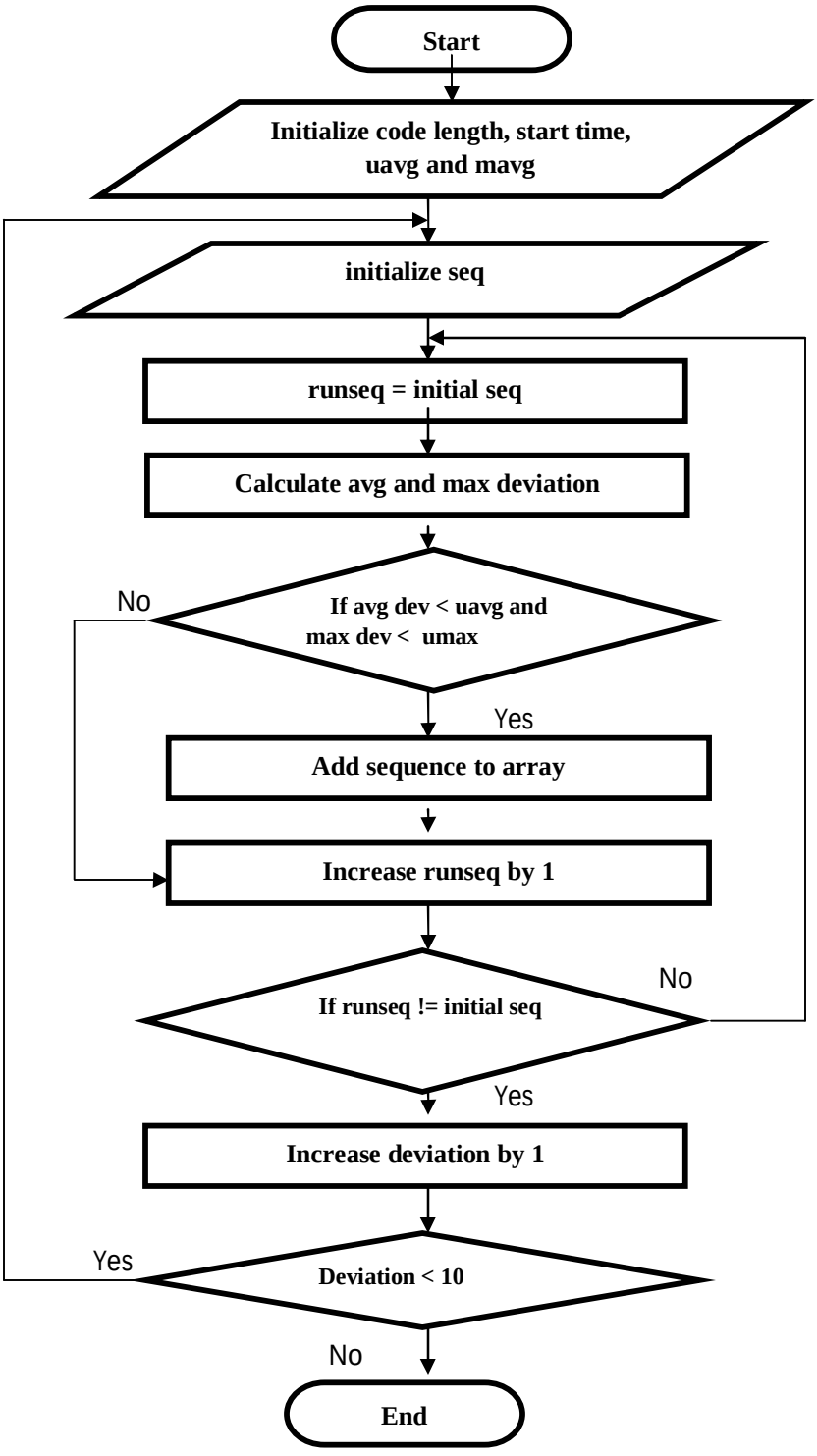

Figure 38: Flow Chart for generating orthogonal sequences

The algorithm used to do the exhaustive search compares all possible sets for orthogonality between the available codes. From the above algorithm it is evident that it searches all possible codes formed by a particular code length. This algorithm is bit exhaustive but gives maximum possible codes available for a particular code.

Definition of Average Deviation and Max Deviation:

Average Deviation: The average deviation from 90 degrees of the angle of any particular signal to all other signals must not exceed a threshold value.

$$\begin{aligned} &Example: Let\ S_1, S_2, S_3, S_4\ and\ S_5\ be\ semi-orthogonal\ codes \\ &\quad Then\ to\ find\ S_6 \\ &\quad Let\ the\ angles\ of\ S_6\ to\ S_i\ be\ A_i\ \ for\ 1 \le i \le 5 \\ &\quad To\ consider\ S_6\ as\ a\ partial\ orthogonal\ signal \\ &\left(\frac{(90^0 - A_1) + (90^0 - A_2) + (90^0 - A_3) + (90^0 - A_4) + (90^0 - A_5)}{5} \right) < threshold\ value \end{aligned}$$

The threshold for average deviation is decided by code length. For larger code lengths the deviation can be smaller to get a good amount of codes. But for smaller code lengths the deviation must be considered high to have a minimum number of codes. So for a code length of 16 the deviation is 8 degrees.

Maximum Deviation: The deviation from 90 degrees of the angle of any particular signal to any other signal must not exceed a maximum deviation. The condition to be followed for S_6 to be semi-orthogonal is shown below.

$$(90 - A_1)\ or\ (90 - A_2)\ or\ (90 - A_3)\ or\ (90 - A_4)\ or\ (90 - A_5) < \max\ deviation$$

In the above algorithm the codes can be generated with respect to different parameters:

- Variable code length with fixed deviation and fixed initial sequence.
- Variable deviation with fixed code length and fixed initial sequence.
- Variable initial sequence with fixed code length and fixed deviation.

4.6.2 Variable Code Length with Fixed Deviation and Fixed Initial Sequence

The algorithm was set to run for different codes with fixed deviation for every code length. This gives us a good idea about how the trend goes between the 2-level and 3-level with respect to deviation and code length.

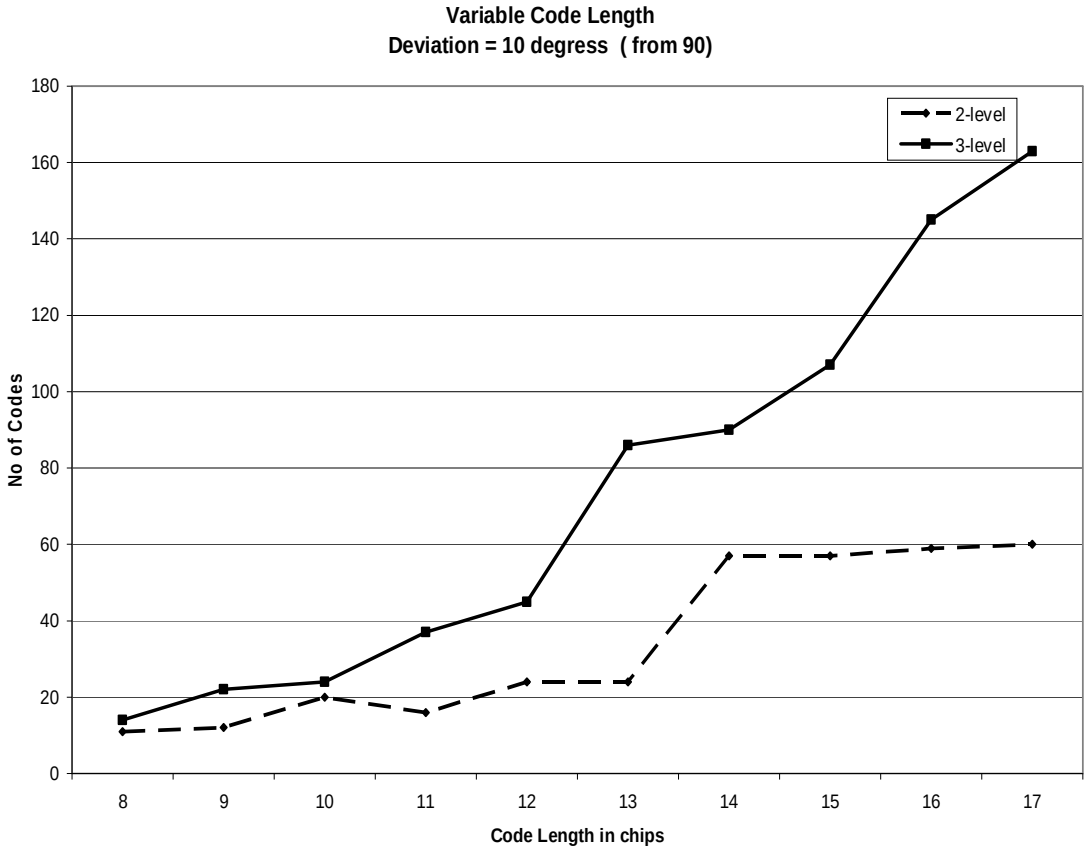

Figure 39: Comparison of codes formed by 2-level and 3-level with varying code length

From Figure 39 we can see that for a low code length, the number of codes generated from 2-level and 3-level are almost same. But when the code length increases the difference gradually increases for 3-level when compared to 2-level. This means that for higher code lengths, even for very low deviation, we can get more number of sets for 3-level than 2-level. The graphs shown above are only for the code word lengths from 8 to 17. This is because, the time consumed by the algorithm to generate the code words are very high. Below is the graph showing time required to generate codes. As the exhaustive search algorithm requires a lot of computing power we have to use super computer to generate the semi-orthogonal codes. For this purpose we got access to Ohio Super Computer (www.osc.edu) for our computation purposes. They have a lot of clusters running different operating systems and capable of running several programming languages including Java. The simulation times shown in Figure 40 are when the algorithm is executed on the Ohio super computer with 4 nodes.

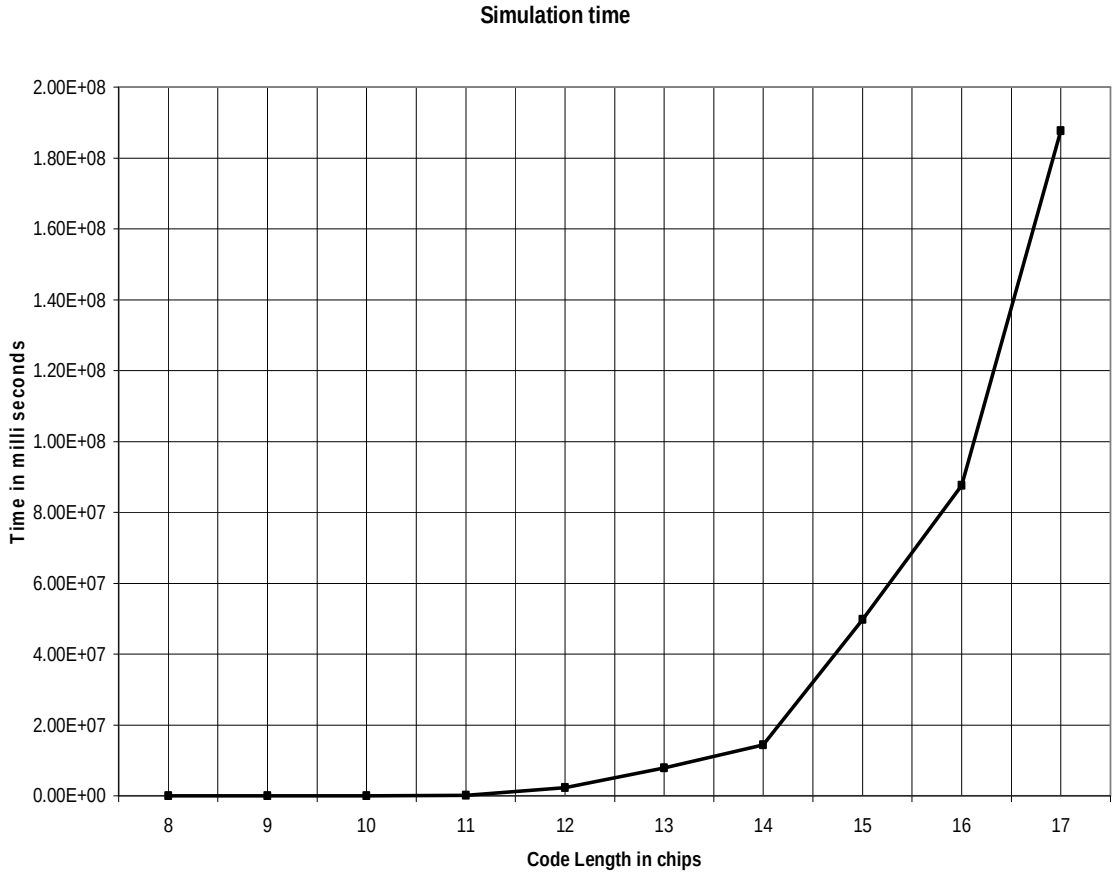

Figure 40: Run time for generating orthogonal codes

From the Figure 40 it is evident that the time to generate the codes increases exponentially. This restricted the algorithm to generate semi orthogonal codes for code lengths less than 18. Even if we optimize the code and further run on more nodes instead of 4 we can at most attain codes for code length of 20.

4.6.3 Variable Deviation with Fixed Code Length and Fixed Initial Sequence

In this case the algorithm is set to run with constant code length (number of chips is set to 17) and the deviation was increased from 0 to 18 degrees from 90^0 (perfectly

orthogonal). This experiment is done to show how the deviation affects the generated codes between 2-level and 3-level. Figure 41 shows the number of codes formed for 2-level and 3-level for deviation ranging from 0 to 18 for a code word of 17.

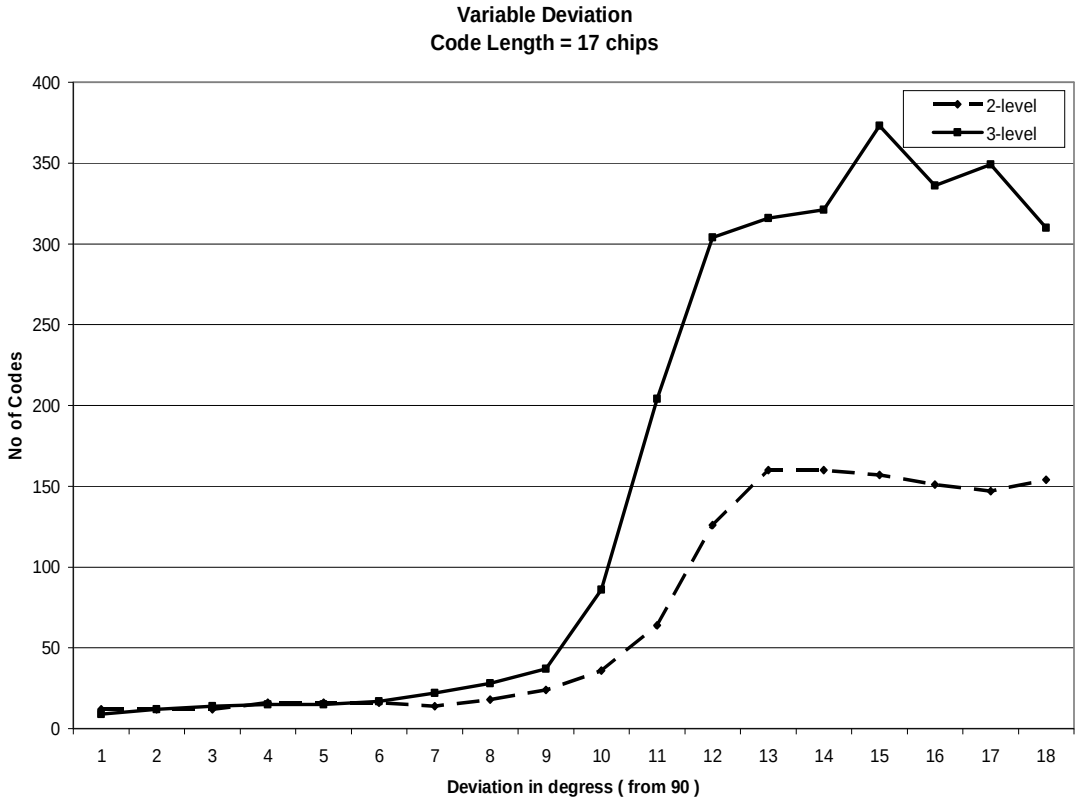

Figure 41: Comparison of codes formed for 2-level and 3-level with varying deviation

In Figure 41 we can observe that at low deviation the set of codes formed are same for both 2-level and 3-level. But as deviation increases the available code words are increasing gradually. In Figure 41 we can observe that up to deviation of about 9 degrees both 2-level and 3-level have approximately the same number of code words formed, but for greater code deviations there are significantly larger sets of code words

for 3-level than 2-level. This proves that for high code length, a small increase in deviation would give rise to large amount of code words.

4.6.4 Variable Initial Sequence with Fixed Code Length and Fixed Deviation

The algorithm given in Figure 38 starts with an initial sequence to be considered. But this initial sequence also affects the overall generated code words. To get exact comparison between 2-level and 3-level we must have maximum possible code words for a particular code length and particular deviation. The graph presented here is for code length equal to 17 and deviation equal to 6 degrees but with different starting sequences.

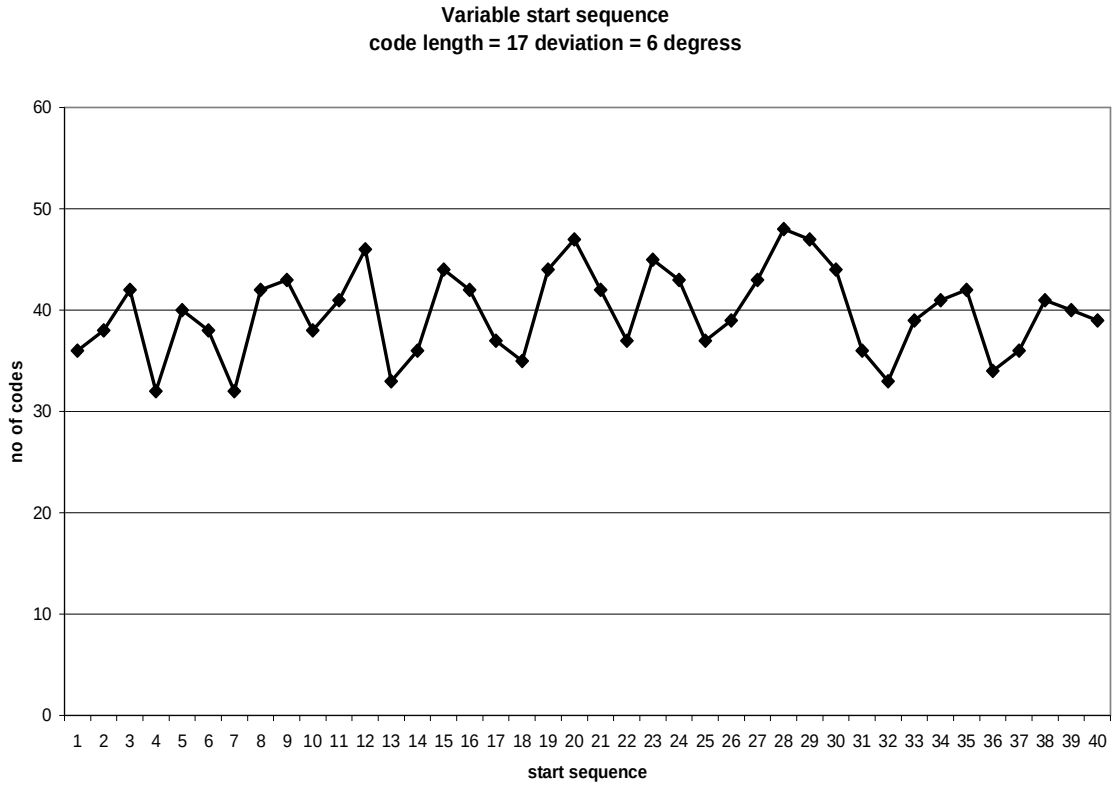

Figure 42: Codes formed by 3-level with variable start sequence

We see in Figure 42 that depending on the initial sequence used, the number of code words generated changes from a minimum of 32 to a maximum of 48, with an approximate average of 40.

4.6.5 Analysis for Larger Code Words

From above experiments it is evident that obtaining semi-orthogonal codes for larger code lengths (typically code length greater than 17) requires a lot of computation resource and time. It is also observed that 3-level code words has more possible codes formed than the 2-level code words for any given code length or deviation and difference in number of code words formed is increasing when we increase code length or deviation. The graph shown in Figure 43 is a three dimensional graph between code length, deviation and number of code words formed for 3-level codes.

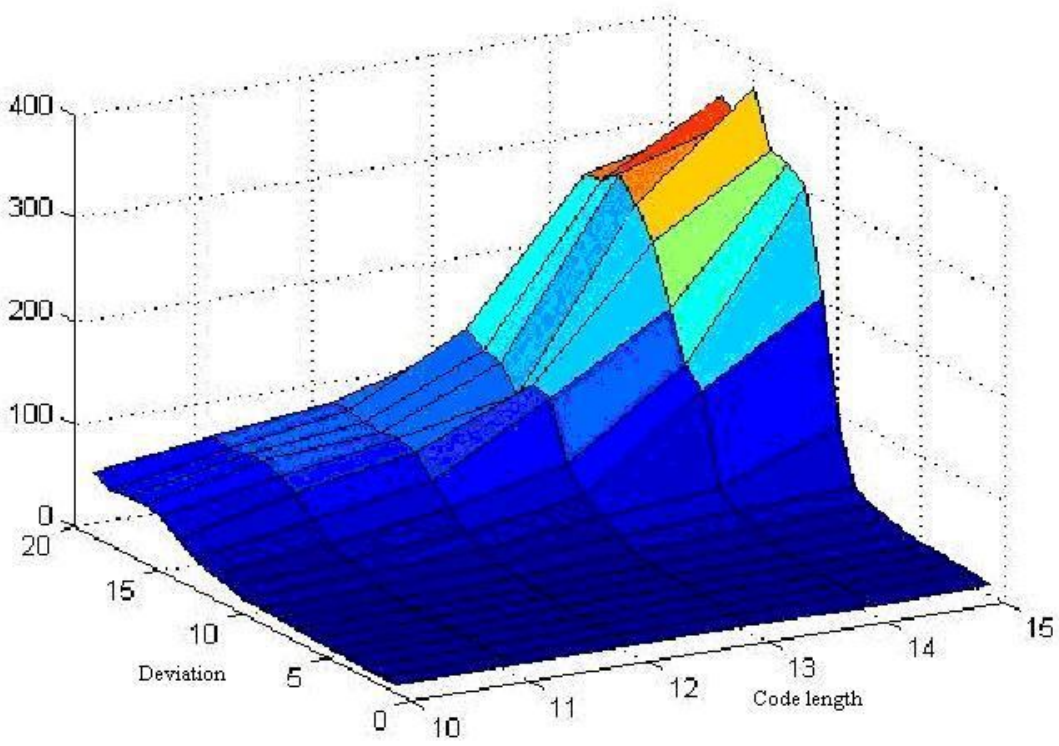

Figure 43: Three-dimensional graph between number of codes and number of deviation

From the Figure 43 we can clearly observe that for larger deviation and larger code length the number of codes formed is increasing exponentially. We can conclude that at large code lengths even for small deviation the number of codes formed are much larger for 3-level than 2-level. As the deviation is same for both of them we may get same multiple access interference but can accommodate a larger number of users. On the other hand for same number of users, we can have less multiple access interference for generalized DSSS than ordinary DSSS as the deviation would be small.

4.7 Conclusion:

We have considered various multiple access scenarios by modulating base signal with random codes, perfectly orthogonal codes and semi-orthogonal codes. For random and perfectly orthogonal codes we have observed that there is no difference in performance between 2-level and 3-level.

We have investigated various ways to generate semi-orthogonal codes. In this thesis we have first attemped to generate semi-orthogonal codes for both 3-level and 2-level using Gold sequences. We have observed that the codes formed by Gold sequences for 3-level do have equal deviation and also investigated the reason behind this failure. The failure of the algorithm is mainly because of the characteristics possessed by m-sequences and preferred sequences of m-sequences for 2-level.

We have also investigated the generation of semi-orthogonal signals by using exhaustive search and proposed an algorithm to effectively generate semi-orthogonal signals. We have generated semi-orthogonal signals by considering various parameters like variable deviation, variable code word, variable levels and variable start sequences.

CHAPTER V

CONCLUSIONS AND FUTURE WORK

Conclusion

This thesis investigates the generation of orthogonal as well as semi orthogonal codes for 2-level and 3-level by using Gold sequence and exhaustive search algorithms. We have constantly observed that 3-level has more codes formed than 2-level for any deviation or code length. We also developed java library for simulating ordinary and generalized direct sequence spread spectrum under Gaussian, multi-path and multiple access interference and compared the performance. Important conclusions and claims are stated below.

Simulation

As a part of the thesis we developed several programs and algorithms to simulate the communication systems. We have developed a rich set of library functions which are very useful to simulate various scenarios for generalized spread spectrum. The library is

well organized in its structure and all the components are designed such that it is loosely coupled with each other and well encapsulated between identical functions which make it easy for other researches to use them and apply to their research. In addition to the scope of this research we have developed java versions of existing algorithms like convolutional coding, equation generator for 3-level m-sequence, convolutional interleaver, etc. The library is well documented using Javadoc which makes the library easy to use by other developers. We have developed utility functions like finding auto correlation, cross correlation between two signals, sampler, adder, multiplier, integrator and BER which are very commonly used by any researcher dealing with simulation of generalized spread spectrum. The library even has sample programs for simulation of few scenarios like Gaussian, multi-path and multiple access which give future researchers a good understanding of their use. As the library is designed for generalized spread spectrum it is more optimized for simulation of generalized spread spectrum even though it can be used to simulate ordinary spread spectrum.

Multi-path Simulation

In the second part of this thesis we have focused more on the simulation of much more complicated scenarios like multi-path fading. We have created a simulation environment where the interference is mainly from multi-paths from a given signal with delay spreads ranging from 0.1 msec to 10 msec and modulating signal frequency varying from 1 MHz to 10 MHz. We have simulated both ordinary and generalized spread spectrum in the above scenarios with random code as the modulating signal and found that the ordinary spread spectrum is performing better than generalized spread spectrum. We have even simulated with m-sequences as modulating signal with various frequencies

and delay spread and found that ordinary spread spectrum is performing better than generalized spread spectrum. We concluded that generalized spread spectrum does not perform better than ordinary spread spectrum when we consider the above scenariso (which somewhat replicates the IS-95 communication model [8]).

Multiple Access Simulation

In this part of the thesis we have concentrated mainly on the effects of generalized DSSS and ordinary DSSS with respect to multiple-access over several users. We have considered three cases which are random signal, perfectly orthogonal signal and semi–orthogonal signal. In the first case we have simulated with a random code assigned to each user where the cross correlation properties are unknown and observed that both ordinary and generalized spread spectrum have equal performance. In the second case we have simulated by assigning each user a unique code word which is perfectly orthogonal and observed that both 2-level and 3-level have equal performance which matches exactly with BPSK theoretical results. This behavior is because for perfectly orthogonal signals there is no multiple-access interference in the system and the only interference in the system is Gaussian noise. For simulating with semi-orthogonal signals there is no simple algorithm to produce semi-orthogonal signals and for this reason we have considered the generation of semi-orthogonal signals for 3-level.

In this part of the thesis we have generated semi-orthogonal and perfectly-orthogonal signals, and observed their behavior over deviation, code length and cross correlation properties. We have investigated the cross correlation properties of the signals formed for 3-level and 2-level by using Gold sequence algorithm and found that

2-level possesses good and even cross correlation between signals compared to 3-level. This is because of auto correlation properties of 3-level m-sequences which have are three valued as shown in Figure 24. The second reason of the failure is due to lack of preferred pair of 3-level m-sequences which is required for generation of Gold sequences.

Since Gold sequences did not produce a good set of semi-orthogonal codes for 3-level, we have developed an algorithm (exhaustive search algorithm) to search for possible semi-orthogonal codes in a set of all possible codes for a given code length. We have observed that the number of semi-orthogonal codes formed for a particular code length and deviation for 3-level is more than 2-level. We have generated semi-orthogonal codes for a fixed code length and variable deviation, and observed that for 3-level the set of semi-orthogonal codes formed are much greater than 2-level. We also generated semi-orthogonal codes for a fixed deviation and variable code length and observed the trend of the codes formed for 3-level and 2-level. From the above two statements we can conclude that for any code length 3-level has a larger set of semi-orthogonal signals. This means that we can accommodate more number of users with same performance if we use 3-level semi orthogonal codes than using 2-level semi orthogonal codes. Even for the same number of users generalized DSSS can have a smaller multiple access interference than ordinary DSSS if the above semi orthogonal codes are assigned to each user in CDMA.

Future Work

There are several aspects that can still be considered for future work as stated below.

- One of the possible ways is to test the multi-path scenario with other available sequences instead of m-sequences and on different frequencies.
- There is also a possibility to work on generating signature sequence which has different auto correlation properties then m-sequences.
- The multi-path scenario considered in this thesis is loosely based on IS-95 CDMA model with 1.222 Mbps as code rate and high frequency rate with 10 Mbps but the data frequency used is same for both scenarios.
- The other aspect is of extending the java library to be a frame work which can simulate all other conditions with filtering effect of the channel.
- Generation of semi-orthogonal signals using advanced search algorithms that can generate up to 64-code length. Now the search used in this thesis uses an exhaustive search that takes a long time to generate a code of 20-bit length.
- The multiple access scenarios can be tested for lengths of codes larger than 20, which were not considered in this thesis.
- Finally all the above scenarios are compared between 2-level and 3-level. This can be extended to inspect for larger number of levels like 5, 7, 9 and higher.

REFERENCES

1. Andrew J. Viterbi, CDMA: Principle of Spread Spectrum Communication. Addison-Wesley Wireless Comunications Series 1995.

2. Murad Hizlan, "A Generalization of Direct-Sequence Spread-Spectrum," submitted to *Journal of the Franklin Institute*, February 2009,

3. Murad Hizlan and Brian L. Hughes, "Worst-Case Error Probability of a Spread Spectrum System in Energy Limited Interference," *IEEE Trans. Comm.*, vol 39, pp. 1193-1196, August 1991.

4. Hariharan Ramaswamy, "Generation of Generalized Signature Sequences," *Masters Thesis*, Cleveland State University, 2004.

5. Brian L. Hughes and Murad Hizlan, "An Asymptotically Optimal Random Modem and Detector for Robust Communication," *IEEE Trans. Inform. Theory*, vol 36, pp. 810-821, July 1990.

6. Manohar Vellala, "Coded Generalized Direct-Sequence Spread-Spectrum with Specific Codes," *Masters Thesis*, Cleveland State University 2004.

7. Madan Venn, "Convolutionally Coded Generalizated Direct Sequence Spread Spectrum," *Masters Thesis*, Cleveland State University, 2008.

8. Robert Clyde Dixon, Spread Spectrum Systems. John Wiley & Sons, Inc 1976.

9. Ranga Kalakuntla, "Further Generalized Direct-Sequence Spread-Spectrum," *Masters Thesis*, Cleveland State University, 2004.

10. Sree Upadhyayula, "An Asymptotic Analysis of the Worst-Case Performance of Coded Generalized Direct Sequence Spread Sequence," *Master Thesis*, Cleveland State University, 2004.

11. Konstantin Matheou, “A Worst-Case Comparision of Generalized and Ordinary DSSS in a Multipath Environment,” *Masters Thesis*, Cleveland State University, 2005.

12. A. J. Paulray and C. B. Papadias, “Space-Time Processing for Wireless Communications,” *Signal Processing Magazine*, November 1997, pp. 49-83.

13. Michael C. Jeruchim, Philip Balaban, and K. Sam Shanmugan, Simulation of Communication Systems, Applications of Communication Theory. Plenum press: 1992.

14. Bernard Skylar, Digital Communications: Fundamentals and Applications. Pearson Education: 2002.

APPENDIX

Appendix A: Gaussian Noise Simulation Code

This appendix provides the simulation program in java for both generalized and ordinary spread spectrum for Gaussian Scenario.

```
/************Program for ordinary Generalized Spread Spectrum in Gaussian Noise*********

/*
 * Main.java
 *
 * Created on July 6, 2007, 1:23 AM
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */
package thesis.legacy;
/**
 *
 * @author indra
 */
import java.util.*;
import java.math.*;
import java.io.*;

public class Main
{
    /** Creates a new instance of Main */
    private static Random randomSequenceGenerator=new Random();
    private static Random randomNoiseGenerator=new Random();
    private static Random randomCodeGenerator=new Random();

    public Main()
    {

    }

    /**
     * @param args the command line arguments
     */
```

```
    public static void main(String[] args)throws IOException
    {
        BufferedReader userIn=new BufferedReader(new
InputStreamReader(System.in));
        System.out.println("Enter the length of the sequence");
        int  dl=Integer.parseInt(userIn.readLine());
        System.out.println("Enter the length of the code");
        int cl=Integer.parseInt(userIn.readLine());
        long nErrors=0;
        long nBits=0;
        for(int snr=0;snr<11;snr++){
            while(nErrors<100){
                int[] intSeqDataArray=gCSeq(dl);
                //print(dataSequence);
                int[][] intSeqCodeArray=gCode(dl,cl);
                // print(codeSequence);
                int[][] negateCodeSequence=negate(intSeqCodeArray);
                // print(negateCodeSequence);
                int[][]
multipliedSequence=multiplier(intSeqDataArray,intSeqCodeArray);
                //System.out.println("\n_____________________");
                // print(multipliedSequence);
                double[]
noiseSequence=noiseGenerator(dl*cl,snr,intSeqCodeArray[0].length);
               // print(noiseSequence);
                double[][]
noisePlusSignal=adder(multipliedSequence,noiseSequence);
                double[]
positiveSequence=integratorPlusMultiplier(noisePlusSignal,intSeqCodeArray);
               // print(positiveSequence);
                double[]
negativeSequence=integratorPlusMultiplier(noisePlusSignal,negateCodeSequence);
               // print(negativeSequence);
                int[]
receivedSequence=comparator(positiveSequence,negativeSequence);
                 //print(receivedSequence);
                nErrors=nErrors+countErrors(intSeqDataArray,receivedSequence);
                nBits=nBits+dl;
            }
            double ber=berCal(nBits,nErrors);
            System.out.println("for the given snr "+(double)snr+" ber is
"+ber);
```

```
            nBits=0;
            nErrors=0;
        }

    }

    private static int[] gCSeq(int length){
        int[] result=new int[length];
        for(int i=0;i<length;i++) {
            if(randomSequenceGenerator.nextInt(100)%2==0) result[i]=-1;
            else result[i]=1;
        }
        return result;
    }

    private static int[][] gCode(int dL,int cL){
        int[][] result=new int[dL][cL];
        for(int i=0;i<dL;i++){
            for(int j=0;j<cL;j++){
                if(randomSequenceGenerator.nextInt(100)%2==0) result[i][j]=-1;
                else result[i][j]=1;
            }
        }
        return result;
    }

    private static int[][] multiplier(int[] seqData,int[][] seqCode){
        int[][] result=new int[seqCode.length][seqCode[0].length];
        for(int i=0;i<seqCode.length;i++){
            for(int j=0;j<seqCode[0].length;j++){
                result[i][j]=seqCode[i][j]*seqData[i];
            }
        }
        return result;
    }

    private static double[] noiseGenerator(int length,double snr,int cL){
        double[] result=new double[length];
        for(int i=0;i<length;i++){
            result[i]=
randomNoiseGenerator.nextGaussian()*Math.sqrt((double)cL/2)/Math.sqrt(Math.pow(
```

```
10,(double)snr/10));
        }
        return result;
    }

    private static double[][] adder(int[][] data,double[] noise){
        double[][] result=new double[data.length][data[0].length];
        int n=0;
        for(int i=0;i<result.length;i++){
            for(int j=0;j<result[0].length;j++){
                result[i][j]=noise[n]+data[i][j];
                n++;
            }
        }
        return result;
    }

    private static double[] integratorPlusMultiplier(double[][] array,int[][]
data){
        double[] result=new double[array.length];
        for(int i=0;i<array.length;i++){
            double temp=0;
            for(int j=0;j<array[0].length;j++){
                //array[i][j]=array[i][j]*data[i][j];
                temp=temp+array[i][j]*data[i][j];
            }
            result[i]=temp;
        }
        return result;
    }

    private static void print(int[] array){
        System.out.println();
        for(int i=0;i<array.length;i++){
            System.out.print(array[i]+"   ");
        }
    }

    private static void print(double[] array){
        System.out.println();
        for(int i=0;i<array.length;i++){
            System.out.print(array[i]+"   ");
```

```
        }
    }

    private static void print(double[][] array){
        System.out.println();
        for(int i=0;i<array.length;i++){
            System.out.println();
            for(int j=0;j<array[0].length;j++)
                System.out.print(array[i][j]+"   ");
        }
    }

    private static void print(int[][] array){
        System.out.println();
        for(int i=0;i<array.length;i++){
            System.out.println();
            for(int j=0;j<array[0].length;j++)
                System.out.print(array[i][j]+"   ");
        }
    }

    private static int[][] negate(int[][] code){
        int[][] result=new int[code.length][code[0].length];
        for(int i=0;i<code.length;i++){
            for(int j=0;j<code[0].length;j++){
                result[i][j]=code[i][j]*-1;
            }
        }
        return result;
    }

    private static int[] comparator(double[] positive,double[] negative){
        int[] result=new int[positive.length];
        for(int i=0;i<result.length;i++){
            if(positive[i]>=negative[i]) result[i]=1;
            else result[i]=-1;
        }
        return result;
    }

    private static long countErrors(int[] originalSequence,int[]
receiveSequence){
```

```
        long result=0;
        for(int i=0;i<originalSequence.length;i++){
            if(originalSequence[i]!=receiveSequence[i]) result++;
        }
        return result;
    }

    private static double berCal(long totalNoOfBits,long totalNoOfErrors){
        double result=0;
        result=(double)totalNoOfErrors/totalNoOfBits;
        return result;
    }
}

/* Code for simulating generalized spread spectrum in Gaussian noise

/*
 * new_3level.java
 *
 *
 * Created on July 9, 2007, 3:49 AM
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

package thesis.legacy;

/**
 *
 * @author indra
 */

import java.util.*;
import java.math.*;
import java.io.*;
import sun.tools.tree.LengthExpression;

public class new_3level {
       /** Creates a new instance of new_3level */
       private static Random randomSequenceGenerator = new Random();
       private static Random gaussianNoiseGenerator = new Random();
       private static Random randomCodeGenerator = new Random();
```

```
	public new_3level() {
	}

	/**
	 * @param args
	 *            the command line arguments
	 */
	public static void main(String[] args) throws IOException {
		BufferedReader userIn = new BufferedReader(new InputStreamReader(
				System.in));
		System.out.println("Enter the length of the sequence");
		int dl = Integer.parseInt(userIn.readLine());
		System.out.println("Enter the length of the code");
		int cl = Integer.parseInt(userIn.readLine());
		long nErrors = 0;
		long nBits = 0;
		double[] snrToWrite = new double[11];
		double[] berToWrite = new double[11];
		for (int snr = 0; snr < 11; snr++) {
			while (nErrors < 100) {
				int[] intSequenceForData = sequenceGenerator(dl);
				System.out.println();
				System.out.println("Data Sequence");
				print(intSequenceForData);
				int[][] sequenceForCode = codeGenerator(dl, cl);
				System.out.println();
				System.out.println("Code word");
				print(sequenceForCode);
				double[][] codeAfterGeneralization =
generalizeSequence(sequenceForCode);
				System.out.println();
				System.out.println("Generalized Sequence");
				print(codeAfterGeneralization);
				// print(negateCodeSequence);
				double[][] codeSequenceAfterGeneralizationNegate =
negate(codeAfterGeneralization);
				double[][] sequenceAfterMultiplication =
multiply(intSequenceForData,
						codeAfterGeneralization);
				// System.out.println("\n_____________________");
				System.out.println();
```

```
                        System.out.println("Multiplied Sequence");
                        print(sequenceAfterMultiplication);
                        double[] doubleSequenceForNoise = generateNoise(dl *
cl,
                                        snr, sequenceForCode[0].length);
                        System.out.println();
                        System.out.println("Noise sequence");
                        printNoise(doubleSequenceForNoise, cl, dl);
                        double[][] sequenceNoisePlusSignal =
addArrays(sequenceAfterMultiplication,
                                        doubleSequenceForNoise);
                        System.out.println();
                        System.out.println("Noise plus signal");
                        print(sequenceNoisePlusSignal);
                        double[] positiveSequence = integrator(
                                        sequenceNoisePlusSignal,
codeAfterGeneralization);
                        System.out.println();
                        System.out.println("Multiplied and integrate
sequence");
                        print(positiveSequence);
                        double[] negativeSequence = integrator(
                                        sequenceNoisePlusSignal,
codeSequenceAfterGeneralizationNegate);
                        // print(negativeSequence);
                        int[] decodedSequence = comparator(positiveSequence,
                                        negativeSequence);
                        System.out.println();
                        System.out.println("Decoded sequence");
                        print(decodedSequence);
                        nErrors = nErrors
                                        + getErrors(intSequenceForData,
decodedSequence);
                        nBits = nBits + dl;
                }
                double ber = calculateBER(nBits, nErrors);
                System.out.println("for the given snr " + (double) snr + "
ber is "
                                + ber);
                snrToWrite[snr] = snr;
                berToWrite[snr] = ber;
                nBits = 0;
```

```
                nErrors = 0;
            }
            System.out.println("Enter the file to be written");
            FileOutputStream fout = new FileOutputStream(userIn.readLine());
            for (int i = 0; i < snrToWrite.length; i++) {
                fout.write(Double.toString(snrToWrite[i]).getBytes());
                fout.write("\t".getBytes());
                fout.write(Double.toString(berToWrite[i]).getBytes());
                fout.write("\n".getBytes());
            }
        }

        private static int[] sequenceGenerator(int length) {
            int[] result = new int[length];
            for (int i = 0; i < length; i++) {
                if (randomSequenceGenerator.nextInt(100) % 2 == 0)
                    result[i] = -1;
                else
                    result[i] = 1;
            }
            return result;
        }

        private static int[][] codeGenerator(int dataLength, int codeLength) {
            int[][] result = new int[dataLength][codeLength];
            for (int i = 0; i < dataLength; i++) {
                for (int j = 0; j < codeLength; j++) {
                    int temp = randomSequenceGenerator.nextInt(100) % 3;
                    if (temp == 0)
                        result[i][j] = 0;
                    else if (temp == 1)
                        result[i][j] = -1;
                    else
                        result[i][j] = 1;
                }
            }
            return result;
        }

        private static int[][] multiply(int[] dataSequence, int[][] codeSequence)
{
            int[][] result = new
```

```
int[codeSequence.length][codeSequence[0].length];
		for (int i = 0; i < codeSequence.length; i++) {
			for (int j = 0; j < codeSequence[0].length; j++) {
				result[i][j] = codeSequence[i][j] * dataSequence[i];
			}
		}
		return result;
	}

	private static double[][] multiply(int[] dataSequence,
			double[][] codeSequence) {
		double[][] result = new
double[codeSequence.length][codeSequence[0].length];
		for (int i = 0; i < codeSequence.length; i++) {
			for (int j = 0; j < codeSequence[0].length; j++) {
				result[i][j] = codeSequence[i][j] * dataSequence[i];
			}
		}
		return result;
	}

	private static double[][] generalizeSequence(int[][] sequence) {
		double[][] result = new
double[sequence.length][sequence[0].length];
		for (int i = 0; i < sequence.length; i++) {
			int noOfZeros = 0;
			for (int j = 0; j < sequence[0].length; j++) {
				if (sequence[i][j] == 0)
					noOfZeros++;
			}
			// System.out.println(noOfZeros);
			int noOfOnes = sequence[0].length - noOfZeros;
			// System.out.println(noOfOnes);
			double value = Math.sqrt((double) sequence[0].length /
noOfOnes);
			double addedValue = value - 1;
			// System.out.println(value);
			for (int j = 0; j < sequence[0].length; j++) {
				if (sequence[i][j] == 1)
					result[i][j] = sequence[i][j] + addedValue;
				else if (sequence[i][j] == -1)
					result[i][j] = sequence[i][j] - addedValue;
```

```
                }
            }
            return result;
        }

        private static double[] generateNoise(int length, double snr, int
codeLength) {
            double[] result = new double[length];
            for (int i = 0; i < length; i++) {
                result[i] = gaussianNoiseGenerator.nextGaussian()
                            * Math.sqrt((double) codeLength / 2)
                            / Math.sqrt(Math.pow(10, (double) snr / 10));
            }
            return result;
        }

        private static double[][] addArrays(int[][] data, double[] noise) {
            double[][] result = new double[data.length][data[0].length];
            int n = 0;
            for (int i = 0; i < result.length; i++) {
                for (int j = 0; j < result[0].length; j++) {
                    result[i][j] = noise[n] + data[i][j];
                    n++;
                }
            }
            return result;
        }

        private static double[][] addArrays(double[][] data, double[] noise) {
            double[][] result = new double[data.length][data[0].length];
            int n = 0;
            for (int i = 0; i < result.length; i++) {
                for (int j = 0; j < result[0].length; j++) {
                    result[i][j] = noise[n] + data[i][j];
                    n++;
                }
            }
            return result;
        }

        private static double[] multiplyAndIntegrate(double[][] array, int[][]
data) {
```

```
        double[] result = new double[array.length];
        for (int i = 0; i < array.length; i++) {
                double temp = 0;
                for (int j = 0; j < array[0].length; j++) {
                        // array[i][j]=array[i][j]*data[i][j];
                        temp = temp + array[i][j] * data[i][j];
                }
                result[i] = temp;
        }
        return result;
}

private static double[] integrator(double[][] array,
                double[][] data) {
        double[] result = new double[array.length];
        for (int i = 0; i < array.length; i++) {
                double temp = 0;
                for (int j = 0; j < array[0].length; j++) {
                        // array[i][j]=array[i][j]*data[i][j];
                        temp = temp + array[i][j] * data[i][j];
                }
                result[i] = temp;
        }
        return result;
}

private static void print(int[] array) {
        System.out.println();
        System.out.print("[");
        for (int i = 0; i < array.length; i++) {
                if (array.length != i + 1) {
                        System.out.print(array[i] + ",\t");
                } else {
                        System.out.print(array[i]);
                }
        }
        System.out.print("]");
}

private static void print(double[] array) {
        System.out.println();
        for (int i = 0; i < array.length; i++) {
```

```
                    Double d = new Double(array[i]);
                    if (d.toString().length() > 5) {
                          System.out.print(d.toString().substring(0, 5) + "
");
                    } else {
                          System.out.print(d.toString() + "   ");
                    }
              }
        }

        private static void print(double[][] array) {
              System.out.println();
              for (int i = 0; i < array.length; i++) {
                    System.out.println();
                    for (int j = 0; j < array[0].length; j++) {
                          Double d = new Double(array[i][j]);
                          if (d.toString().length() > 5) {
                                System.out.print(d.toString().substring(0, 5)
+ "   ");
                          } else {
                                System.out.print(d.toString() + "   ");
                          }
                    }
              }
        }

        public static void printNoise(double[] array, int codeLength, int
dataLength) {
              System.out.println();
              int n = 0;
              for (int i = 0; i < dataLength; i++) {
                    System.out.println();
                    for (int j = 0; j < codeLength; j++) {
                          Double d = new Double(array[n]);
                          n++;
                          if (d.toString().length() > 5) {
                                System.out.print(d.toString().substring(0, 5)
+ "   ");
                          } else {
                                System.out.print(d.toString() + "   ");
                          }
                    }
```

```
	}
}

private static void print(int[][] array) {
	System.out.println();
	for (int i = 0; i < array.length; i++) {
		System.out.println();
		for (int j = 0; j < array[0].length; j++)
			System.out.print(array[i][j] + "   ");
	}
}

private static int[][] negate(int[][] code) {
	int[][] result = new int[code.length][code[0].length];
	for (int i = 0; i < code.length; i++) {
		for (int j = 0; j < code[0].length; j++) {
			result[i][j] = code[i][j] * -1;
		}
	}
	return result;
}

private static double[][] negate(double[][] code) {
	double[][] result = new double[code.length][code[0].length];
	for (int i = 0; i < code.length; i++) {
		for (int j = 0; j < code[0].length; j++) {
			result[i][j] = code[i][j] * -1;
		}
	}
	return result;
}

private static int[] comparator(double[] positive, double[] negative) {
	int[] result = new int[positive.length];
	for (int i = 0; i < result.length; i++) {
		if (positive[i] >= negative[i])
			result[i] = 1;
		else
			result[i] = -1;
	}
	return result;
}
```

```
	private static long getErrors(int[] originalSequence, int[]
receiveSequence) {
		long result = 0;
		for (int i = 0; i < originalSequence.length; i++) {
			if (originalSequence[i] != receiveSequence[i])
				result++;
		}
		return result;
	}

	private static double calculateBER(long totalNoOfBits, long
totalNoOfErrors) {
		double result = 0;
		result = (double) totalNoOfErrors / totalNoOfBits;
		return result;
	}
}
```

Appendix B: Generation of Semi orthogonal Codes

This section contains program for exhaustive search algorithm for both 2-level and 3-level and Gold Sequence Generator for 2-level and 3-level

```
/** Exhaustive Search Algorithm**/


/*
  * new_ortho_3set.java
  *
  * Created on July 26, 2007, 1:23 AM
  *
  * To change this template, choose Tools | Template Manager
  * and open the template in the editor.
  */
 package thesis.legacy;


 /**
  *
  * @author indra
  */
 import java.util.*;
 import java.io.*;

 public class new_ortho_3set {
       /** Creates a new instance of new_ortho_3set */
       public new_ortho_3set() {
       }

       public static void main(String[] args) throws IOException {
              BufferedReader userIn = new BufferedReader(new InputStreamReader(
                           System.in));
              // System.out.println("Enter the length of the code");
              // int codeLength=Integer.parseInt(userIn.readLine());
              // System.out.println("Enter the max allowable deviation");
```

```
            // int maxDeviation=Integer.parseInt(userIn.readLine());
            for (int codeLength = 8; codeLength < 20; codeLength++) {
                  long startTime = System.currentTimeMillis();
                  for (int maxDeviation = 1; maxDeviation < 20;
maxDeviation++) {
                        int[] arr = new int[codeLength];
                        for (int i = 0; i < arr.length; i++) {
                              if (i % 2 == 0)
                                    arr[i] = -1;
                              else
                                    arr[i] = 1;
                        }
                        Vector orthogonal_vector = new Vector();
                        orthogonal_vector.add(arr);
                        orthogonal_vector.trimToSize();
                        for (int i = 0; i < Math.pow(3, codeLength); i++) {
                              double totalDeviation = 0;
                              double avgDeviation = 0;
                              int count = 0;
                              for (int j = 0; j <
orthogonal_vector.capacity(); j++) {
                                    double angle =
getAngle(increasePower(arr),
                                                increasePower((int[])
orthogonal_vector

      .elementAt(j)));
                                    double deviation = 0;
                                    if (angle >= 90)
                                          deviation = angle - 90;
                                    else
                                          deviation = 90 - angle;
                                    if (deviation <= 30)
                                          count++;
                                    totalDeviation = totalDeviation +
deviation;
                              }
                              avgDeviation = totalDeviation
                                          / orthogonal_vector.capacity();
                              if (count == orthogonal_vector.capacity()
                                          && avgDeviation <= maxDeviation)
{
```

```
                                        orthogonal_vector.add(arr);
                                        orthogonal_vector.trimToSize();
                                }
                                arr = changeArray(arr);
                        }
                        // System.out.println(orthogonal_vector.capacity());
                        if (orthogonal_vector.capacity() > 200) {
                                break;
                        }
                        for (int i = 0; i < orthogonal_vector.capacity();
i++) {
                                //
print((int[])orthogonal_vector.elementAt(i));
                        }
                }
                System.out.println(System.currentTimeMillis() - startTime);
        }
    }

    private static int noOfZeros(int[] arr) {
        int noOfZero = 0;
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == 0)
                noOfZero++;
        }
        return noOfZero;
    }

    private static int[] changeArray(int[] startArray) {
        int[] result = new int[startArray.length];
        boolean changeBit = true;
        boolean masterBit = false;
        for (int i = startArray.length - 1; i >= 0; i--) {
            // System.out.println(i);
            if (changeBit) {
                if (startArray[i] == 1)
                    result[i] = 0;
                if (startArray[i] == 0)
                    result[i] = -1;
                if (startArray[i] == -1)
                    result[i] = 1;
                // result[i]=-startArray[i];
```

```
			} else {
				result[i] = startArray[i];
				masterBit = true;
			}
			if (startArray[i] >= 0)
				changeBit = false;
			else
				changeBit = true;
			if (masterBit)
				changeBit = false;
		}
		return result;
	}

	private static void print(int[] array) {
		System.out.println();
		for (int i = 0; i < array.length; i++) {
			System.out.print(array[i] + "   ");
		}
		System.out.println();
	}

	private static void print(double[] array) {
		System.out.println();
		for (int i = 0; i < array.length; i++) {
			System.out.print(array[i] + "   ");
		}
		System.out.println();
	}

	private static double dotProduct(int[] arr1, double[] arr2) {
		double result = 0;
		for (int i = 0; i < arr1.length; i++) {
			result = result + arr1[i] * arr2[i];
		}
		return result;
	}

	private static double dotProduct(double[] arr1, double[] arr2) {
		double result = 0;
		for (int i = 0; i < arr1.length; i++) {
			result = result + arr1[i] * arr2[i];
```

```
	}
	return result;
}

private static double[] increasePower(int[] sequence) {
	double[] result = new double[sequence.length];
	int noOfZeros = 0;
	for (int j = 0; j < sequence.length; j++) {
		if (sequence[j] == 0)
			noOfZeros++;
	}
	// System.out.println(noOfZeros);
	int noOfOnes = sequence.length - noOfZeros;
	// System.out.println(noOfOnes);
	double value = Math.sqrt((double) sequence.length / noOfOnes);
	double addedValue = value - 1;
	// System.out.println(value);
	for (int j = 0; j < sequence.length; j++) {
		if (sequence[j] == 1)
			result[j] = sequence[j] + addedValue;
		else if (sequence[j] == -1)
			result[j] = sequence[j] - addedValue;
	}
	return result;
}

private static double magnitude(double[] arr) {
	double result = 0;
	for (int i = 0; i < arr.length; i++) {
		result = arr[i] * arr[i] + result;
	}
	return Math.sqrt(result);
}

private static double magnitude(int[] arr) {
	double result = 0;
	for (int i = 0; i < arr.length; i++) {
		result = arr[i] * arr[i] + result;
	}
	return Math.sqrt(result);
}
```

```
	private static double getAngle(int[] arr1, double[] arr2) {
		double angle = 0;
		// System.out.println(dotProduct(arr1,arr2));
		angle = Math.acos(dotProduct(arr1, arr2)
				/ (magnitude(arr1) * magnitude(arr2)))
				* 90 / Math.acos(0);
		// System.out.println(angle);
		return angle;
	}

	private static double getAngle(double[] arr1, double[] arr2) {
		double angle = 0;
		// System.out.println(dotProduct(arr1,arr2));
		angle = Math.acos(dotProduct(arr1, arr2)
				/ (magnitude(arr1) * magnitude(arr2)))
				* 90 / Math.acos(0);
		// System.out.println(angle);
		return angle;
	}
}
/** Code For Generating 2-level semi orthogonal signals**/

/*
 * new_orthogonal.java
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

package thesis.legacy;

/**
 *
 * @author indra
 */
import java.io.*;
import java.util.*;
public class new_orthogonal {

    /** Creates a new instance of new_orthogonal */
    public new_orthogonal() {
    }
```

```
     public static void main(String[] args)throws IOException{

         BufferedReader userIn=new BufferedReader(new
InputStreamReader(System.in));

         System.out.println("Enter the length of the code");

         //int codeLength=Integer.parseInt(userIn.readLine());

         //System.out.println("Enter the max allowable deviation");

         //int maxDeviation=Integer.parseInt(userIn.readLine());
         for(int codeLength=10;codeLength<20;codeLength++){
         System.out.println("For the code Length of "+codeLength);
         for(int maxDeviation=1;maxDeviation<20;maxDeviation++)
         {
         int[] arr=new int[codeLength];
         for(int i=0;i<arr.length;i++){
             arr[i]=1;
         }
         Vector orthogonal_vector=new Vector();
         orthogonal_vector.add(arr);
         int[] temp=arr;
         orthogonal_vector.trimToSize();
         for(int i=0;i<Math.pow(2,codeLength);i++){
             double totalDeviation=0;
             double avgDeviation=0;
             int count=0;
             for(int j=0;j<orthogonal_vector.capacity();j++){
                 double
angle=getAngle(arr,(int[])orthogonal_vector.elementAt(j));
                 double deviation=0;
                 if(angle>=90) deviation=angle-90;
                 else deviation=90-angle;
                 if(deviation<=30) count++;
                 totalDeviation=totalDeviation+deviation;
             }
             avgDeviation=totalDeviation/orthogonal_vector.capacity();
            // System.out.println(avgDeviation);

if(count==orthogonal_vector.capacity()&&avgDeviation<=maxDeviation){
                 orthogonal_vector.add(arr);
```

```
          //  print(arr);
            orthogonal_vector.trimToSize();
            i=0;
            arr=temp;
        }
        arr=changeArray(arr);
    }
    System.out.println(orthogonal_vector.capacity());
    if(orthogonal_vector.capacity()>300){
        break;
    }
 /*  for(int i=0;i<orthogonal_vector.capacity();i++){
        print((int[])orthogonal_vector.elementAt(i));

    }*/
    }
    }
}

private static int[] changeArray(int[] startArray){
    int[] result=new int[startArray.length];
    boolean changeBit=true;
    boolean masterBit=false;
    for(int i=startArray.length-1;i>=0;i--){
       // System.out.println(i);
        if(changeBit) result[i]=-startArray[i];
        else
        {
            result[i]=startArray[i];
            masterBit=true;
        }
        if(startArray[i]>=0) changeBit=false;
        else changeBit=true;
        if(masterBit) changeBit=false;
    }
    return result;
   }

private static void print(int[] array){
    System.out.println();
    for(int i=0;i<array.length;i++){
        System.out.print(array[i]+"   ");
```

```
        }
        System.out.println();
    }
    private static int dotProduct(int[] arr1 ,int[] arr2){
        int result=0;
        for(int i=0;i<arr1.length;i++){
            result=result+arr1[i]*arr2[i];
        }
        return result;
    }

    private static double magnitude(int[] arr){
        double result=0;
        for(int i=0;i<arr.length;i++){
            result=arr[i]*arr[i]+result;
        }
        return Math.sqrt(result);
    }

    private static double getAngle(int[] arr1,int[] arr2){
        double angle=0;

angle=Math.acos((double)dotProduct(arr1,arr2)/(magnitude(arr1)*magnitude(arr2))
)*90/Math.acos(0);
      //System.out.println(angle);
      return angle;
    }
 }

/**Code for generating m-sequences and Gold Sequences**/
package thesis.generator;
import thesis.legacy.AutoCorrelation;
import thesis.util.Print;
import thesis.util.Shifter;
import thesis.util.XOR;

public class GoldSequence
{
      public static void main(String[] args)
      {
            //int[] level2Result = generate2levelGoldCode();
            int[][] level3Result = generate3levelGoldCode();
```

```
            double[][] result =
AutoCorrelation.getCrossCorrelationAngleMatrix(level3Result);
            Print.print(result);
      }

      public static int[] generate2levelGoldCode()
      {
            MSequencea_2level mSeq=new MSequencea_2level();
       int[] offSet={1,0,1,1};
       int[] equation={1,0,0,1,1};
       mSeq.setOffSet(offSet);
       mSeq.setEquation(equation);
       mSeq.generateSequence();
       int[]
result=mSeq.changeToSignal(mSeq.generateSequenceToLength((int)(Math.pow(2,offSe
t.length)-1)));
        Print.print(result);
            return result;
      }

      public static int[][] generate3levelGoldCode()
      {
             MSequences_3level mSeq1=new MSequences_3level();
            int[] offSet={1,2,1};
            int[] equation={1,1,0,2};
            mSeq1.setOffSet(offSet);
            mSeq1.setEquation(equation);
            mSeq1.generateSequence();
            int[] result1InCode=mSeq1.generatedSequence;
            Print.print(result1InCode);
            int[] result1InSignal=mSeq1.changeToSignal(result1InCode);
            Print.print(result1InSignal);
            AutoCorrelation aCor=new AutoCorrelation();
            double[]
result1Correlation=aCor.getAutoCorrelation(result1InSignal,1);
            Print.print(result1Correlation);
            MSequences_3level mSeq2=new MSequences_3level();
            int[] offSet1={1,1,1};
            int[] equation1={1,0,1,2};
            mSeq2.setOffSet(offSet1);
            mSeq2.setEquation(equation1);
            mSeq2.generateSequence();
```

```
            int[] result2InCode=mSeq2.generatedSequence;
            Print.print(result2InCode);
            int[] result2InSignal=mSeq1.changeToSignal(result2InCode);
            Print.print(result2InSignal);
            double[]
result2Correlation=aCor.getAutoCorrelation(result2InSignal,1);
            Print.print(result2Correlation);
            //Get the preferred sequences
            double[] crossCorrelation =
aCor.getCrossCorrelation(result1InSignal, result2InSignal, 1);
            Print.print(crossCorrelation);
            // Start gold sequence algorithm
            int[][] goldSequenceArray = new
int[result1InSignal.length*2][result1InSignal.length];
            //Set first Generator with result1InSignal
            //Set Second Generator with result2InSignal
            int[] firstGenerator = result1InCode;
            int[] secondGenerator = result2InCode;
            for(int i=0;i<goldSequenceArray.length/2;i++)
            {
                  goldSequenceArray[i] = XOR.XORFor3level(firstGenerator,
secondGenerator);
                  secondGenerator = Shifter.rightShift(secondGenerator, 1);
            }
            //Set the first Generator with 0
            firstGenerator = new int[result1InCode.length];
            for(int
i=goldSequenceArray.length/2;i<goldSequenceArray.length;i++)
            {
               int[] tempArray=XOR.XORFor3level(firstGenerator,
secondGenerator);
               goldSequenceArray[i] = mSeq1.changeToSignal(tempArray);
               secondGenerator = Shifter.rightShift(secondGenerator, 1);
            }
               return goldSequenceArray;
      }
}


package thesis.generator;
import thesis.legacy.AutoCorrelation;
import thesis.legacy.MSequence_2level;
```

```
import thesis.util.Print;

public class MSequencea_2level
{
    /** Creates a new instance of MSequence_2level */
    private int[] present_memory_elements=null;
    private int[] generatedSequence=null;
    private int[] equation=null;
    private int presentIndex=0;

    public static void main(String[] args)
    {
        MSequencea_2level mSeq=new MSequencea_2level();
        int[] offSet={1,0,1,1};
        int[] equation={1,0,0,1,1};
        mSeq.setOffSet(offSet);
        mSeq.setEquation(equation);
        mSeq.generateSequence();
        int[]
result=mSeq.changeToSignal(mSeq.generateSequenceToLength((int)(Math.pow(2,offSe
t.length)-1)));
        Print.print(result);
        AutoCorrelation aCor=new AutoCorrelation();
        double[] correlation=aCor.getAutoCorrelation(result,10);
        System.out.println("\n\nAuto correlation from delay starting from 0 to
"+correlation.length);
        Print.print(correlation);
    }

    public int[] changeToSignal(int[] arr)
    {
        int[] result=new int[arr.length];
        for(int i=0;i<result.length;i++)
        {
            if(arr[i]==0) result[i]=-1;
            else result[i]=1;
        }
        return result;
    }

    public void setOffSet(int[] arr)
    {
```

```
        present_memory_elements=arr;
    }

    public void setEquation(int[] arr)
    {
        equation=arr;
    }

    public void generateSequence()
    {
        generatedSequence=new
int[(int)Math.pow(2,present_memory_elements.length)-1];
        for(int i=0;i<generatedSequence.length;i++)
        {
            generatedSequence[i]=getNextBit();
        }
    }

    public int getNextBit()
    {
        int tempResult=0;
        for(int i=equation.length-1;i>0;i--)
        {
            tempResult=(present_memory_elements[i-1]*equation[i]+tempResult)%2;
        }
        int result=present_memory_elements[present_memory_elements.length-1];
        int[] new_memory_elements=new int[present_memory_elements.length];
        new_memory_elements[0]=tempResult;
        for(int i=0;i<new_memory_elements.length-1;i++)
        {
            new_memory_elements[i+1]=present_memory_elements[i];
        }
        present_memory_elements=new_memory_elements;
        return result;
    }

    public int[] generateSequenceToLength(int length){
        int[] result=new int[length];
        for(int i=0;i<result.length;i++){
            result[i]=generatedSequence[presentIndex%generatedSequence.length];
            presentIndex++;
        }
```

```
        return result;
    }
}


package thesis.generator;
import thesis.legacy.AutoCorrelation;
public class MSequences_3level
{
   /** Creates a new instance of MSequence_3level */
   private int[] present_memory_elements=null;
   public int[] generatedSequence=null;
   private int[] equation=null;
   private int presentIndex=0;

   public static void main(String[] args)
   {
       MSequences_3level mSeq=new MSequences_3level();
       int[] offSet={1,2,1};
       int[] equation={1,1,0,2};
       mSeq.setOffSet(offSet);
       mSeq.setEquation(equation);
       mSeq.generateSequence();
       int[] result=mSeq.generatedSequence;
       print(result);
       int[] result1=mSeq.changeToSignal(result);
       print(result1);
       AutoCorrelation aCor=new AutoCorrelation();
       double[] correlation=aCor.getAutoCorrelation(result1,10);
       print(correlation);
       //System.out.println(correlation.length);
   }

    private static void print(int[] array)
    {
       System.out.println();
       for(int i=0;i<array.length;i++)
       {
           System.out.println(array[i]+"   ");
       }
    }
```

```
    private static void print(double[] array)
    {
       System.out.println();
       for(int i=0;i<array.length;i++){
           System.out.println(array[i]+"   ");
    }
   }

   public  int[] changeToSignal(int[] arr)
   {
       int[] result=new int[arr.length];
       for(int i=0;i<result.length;i++)
       {
          if(arr[i]==2)
                 result[i]=-1;
          else
               result[i]=arr[i];
       }
       return result;
   }

   public void setOffSet(int[] arr)
   {
       present_memory_elements=arr;
   }

   public void setEquation(int[] arr)
   {
       equation=arr;
   }

   public void generateSequence()
   {
       generatedSequence=new
int[(int)Math.pow(3,present_memory_elements.length)-1];
       for(int i=0;i<generatedSequence.length;i++)
       {
           generatedSequence[i]=getNextBit();
       }
   }

   public int getNextBit()
```

```
    {
        int tempResult=0;
        for(int i=equation.length-1;i>0;i--){
            tempResult=(present_memory_elements[i-1]*equation[i]+tempResult)%3;
        }
        int result=present_memory_elements[present_memory_elements.length-1];
        int[] new_memory_elements=new int[present_memory_elements.length];
        new_memory_elements[0]=tempResult;
        for(int i=0;i<new_memory_elements.length-1;i++){
            new_memory_elements[i+1]=present_memory_elements[i];
        }
        present_memory_elements=new_memory_elements;
        return result;
    }

    public int[] generateSequenceToLength(int length)
    {
        int[] result=new int[length];
        for(int i=0;i<result.length;i++)
        {
            result[i]=generatedSequence[presentIndex%generatedSequence.length];
            presentIndex++;
        }
        return result;
    }
}


/** code for Auto correlation and cross correlation functions**/


/*
 * AutoCorrelation.java
 *
 * Created on September 8, 2007, 12:55 PM
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */
package thesis.legacy;
/**
 *
```

```
 * @author indra
 */
public class AutoCorrelation
{
    /** Creates a new instance of AutoCorrelation */
    public AutoCorrelation()
    {
    }

    public static void main(String [] args)
    {
        int[] arr={-1,1,1,-1,1,-1,-1};
        double[] result=getAutoCorrelation(arr,1);
        print(result);
    }

     private static void print(double[] array)
     {
        System.out.println();
        for(int i=0;i<array.length;i++)
        {
            System.out.println(array[i]);
        }
    }

    public static double[] getAutoCorrelation(int[] arr,int precision)
    {
        double[] result=new double[arr.length*precision];
        int[] arr1=new int[result.length];
        int n=0;
        for(int i=0;i<arr.length;i++)
        {
            for(int j=0;j<precision;j++)
            {
                arr1[n]=arr[i];
                n++;
            }
        }
        int[] firstArray=arr1;
        int[] secondArray=arr1;
        for(int i=0;i<result.length;i++){
```

```
result[i]=(double)getCorrelation(firstArray,secondArray)/firstArray.length;
           secondArray=rightShift(secondArray);
         }
         return result;
    }

    public static double[][] getCrossCorrelationMatrix(int[][] arr)
    {
       double[][] result = new double[arr.length][arr.length];
       for(int i=0;i<arr.length;i++){
              for(int j=0;j<arr.length;j++){
                     result[i][j] = (double)getCorrelation(arr[i],
arr[j])/arr[i].length;
              }
       }
       return result;
    }

   public static double[][] getCrossCorrelationAngleMatrix(int[][] arr)
   {
       double[][] result = new double[arr.length][arr.length];
       for(int i=0;i<arr.length;i++)
       {
              for(int j=0;j<arr.length;j++)
              {
                           result[i][j] = getAngle(arr[i], arr[j]);
              }
       }
       return result;
    }

   public static double[] getCrossCorrelation(int[] arr1,int[] arr2,int
precision)
   {
          double[] result=new double[arr1.length*precision];
       int[] arr11=new int[result.length];
       int[] arr21 = new int[result.length];
       int n=0;
       for(int i=0;i<arr1.length;i++){
           for(int j=0;j<precision;j++){
               arr11[n]=arr1[i];
               arr21[n] = arr2[i];
```

```
                n++;
            }
        }
        int[] firstArray=arr11;
        int[] secondArray=arr21;
        for(int i=0;i<result.length;i++){

result[i]=(double)getCorrelation(firstArray,secondArray)/firstArray.length;
           secondArray=rightShift(secondArray);
        }
        return result;
    }

    public static int getCorrelation(int[] arr1,int[] arr2){
         int result=0;
         for(int i=0;i<arr1.length;i++){
             result=result+arr1[i]*arr2[i];
         }
         return result;
     }

     public static double[] getAutoCorrelation(double[] arr)
     {
         double[] result=new double[arr.length];
         double[] firstArray=arr;
         double[] secondArray=arr;
         for(int i=0;i<result.length;i++){
            result[i]=getCorrelation(firstArray,secondArray);
            secondArray=rightShift(secondArray);
         }
         return result;
     }

     public static double getCorrelation(double[] arr1,double[] arr2){
         double result=0;
         for(int i=0;i<arr1.length;i++){
             result=result+arr1[i]*arr2[i];
         }
         return result;
     }

     public static int[] rightShift(int[] arr)
```

```
{
    int[] result=new int[arr.length];
    result[0]=arr[arr.length-1];
    for(int i=0;i<result.length-1;i++){
        result[i+1]=arr[i];
    }
    return result;
}

public static double[] rightShift(double[] arr)
{
    double[] result=new double[arr.length];
    result[0]=arr[arr.length-1];
    for(int i=0;i<result.length-1;i++){
        result[i+1]=arr[i];
    }
    return result;
}

public static int[] leftShift(int[] arr)
{
   int[] result=new int[arr.length];
    result[arr.length-1]=arr[0];
    for(int i=0;i<result.length-1;i++){
        result[i]=arr[i+1];
    }
    return result;
}

public static double[] leftShift(double[] arr){
  double[] result=new double[arr.length];
    result[arr.length-1]=arr[0];
    for(int i=0;i<result.length-1;i++){
        result[i]=arr[i+1];
    }
    return result;
}

private static double getAngle(int[] arr1,int[] arr2){
    double angle=0;
    //System.out.println(dotProduct(arr1,arr2));
```

```
angle=Math.acos(dotProduct(arr1,arr2)/(magnitude(arr1)*magnitude(arr2)))*90/Mat
h.acos(0);
      //System.out.println(angle);
      return angle*2;
    }

    private static double magnitude(double[] arr){
        double result=0;
        for(int i=0;i<arr.length;i++){
            result=arr[i]*arr[i]+result;
        }
        return Math.sqrt(result);
    }

    private static double magnitude(int[] arr)
    {
        double result=0;
        for(int i=0;i<arr.length;i++){
            result=arr[i]*arr[i]+result;
        }
        return Math.sqrt(result);
    }

    private static double dotProduct(int[] arr1 ,int[] arr2)
    {
        double result=0;
        for(int i=0;i<arr1.length;i++){
            result=result+arr1[i]*arr2[i];
        }
        return result;
    }

   private static double dotProduct(double[] arr1 ,double[] arr2)
   {
        double result=0;
        for(int i=0;i<arr1.length;i++){
            result=result+arr1[i]*arr2[i];
        }
        return result;
    }
}
```