IMPLEMENTATION OF TURBO CODES ON GNU RADIO

Mahendra Talasila

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

December 2010

APPROVED:

Shengli Fu, Major Professor and Graduate Program Coordinator
Murali R. Varanasi, Committee Member and Chair of the Department of Electrical Engineering
Yan Wan, Committee Member
Costas Tsatsoulis, Dean of College of Engineering
James D. Meernik, Acting Dean of the Robert B. Toulouse School of Graduate Studies
Talasila, Mahendra. Implementation of turbo codes on GNU radio. Master of Science (Electrical Engineering), December 2010, 67 pp., 34 figures, 2 tables, bibliography, 20 titles.

This thesis investigates the design and implementation of turbo codes over the GNU radio. The turbo codes is a class of iterative channel codes which demonstrates strong capability for error correction. A software defined radio (SDR) is a communication system which can implement different modulation schemes and tune to any frequency band by means of software that can control the programmable hardware. SDR utilizes the general purpose computer to perform certain signal processing techniques. We implement a turbo coding system using the Universal Software Radio Peripheral (USRP), a widely used SDR platform from Ettus. Detail configuration and performance comparison are also provided in this research.
ACKNOWLEDGMENTS

First I would like to express my gratitude to Dr. Shengli Fu, for introducing me to the interesting world of GNU radio and encouraging me to pursue my thesis on this topic and for his guidance throughout my stay in the university. Next, I would like to thank Dr. Murali Varanasi and Dr. Yan Wan for their support as committee members of my thesis. I would also like to thank Eric Blossom for providing tutorials to understand the GNU radio programming. I would also like to thank Matthew C. Valenti and Hamid R. Sadjadpour for providing complete details on how to perform turbo coding. Lastly, I am grateful to my family and friends for their support over the years, and especially during the last several months of intense work on this thesis.
CONTENTS

LIST OF FIGURES vii

CHAPTER 1. INTRODUCTION 1
  1.1. Channel Coding 1
  1.2. Software Defined Radio(SDR) 2
  1.3. Realizable SDR 3

CHAPTER 2. TURBO CODES 5
  2.1. Introduction 5
  2.2. Turbo Encoding 5
    2.2.1. Recursive Encoding 7
    2.2.2. Pseudo Random Interleaving 7
  2.3. Turbo Decoding 7
    2.3.1. MAP Algorithm 9
    2.3.2. Max-Log-MAP Algorithm 11
  2.4. Simulated Results 13
  2.5. Performance Factors 13
  2.6. Selective Serial Concatenation of Turbo Codes 14
    2.6.1. Results 14

CHAPTER 3. GNU RADIO 16
  3.1. Introduction 16
  3.2. USRP Board 17
3.2.1. ADC/DAC Converters

3.2.2. Daughter Boards

3.2.3. FPGA

3.2.4. USRP Transmit and Receive paths

3.3. GNU Radio Installation

3.3.1. Installing GNU Radio

3.4. Dial-tone Program

CHAPTER 4. BPSK MODULATION

4.1. Introduction

4.2. BPSK Modulation

4.2.1. System Model

4.2.2. Implementation

4.2.3. Transmission of Data using BPSK

4.3. BPSK Demodulation

4.3.1. System Model

4.3.2. Implementation

4.3.3. Receiver

4.4. Results

CHAPTER 5. TURBO CODES IMPLEMENTATION

5.1. Introduction

5.2. Implementation

5.2.1. Transmitter

5.2.2. Receiver

5.3. Results

CHAPTER 6. DEMO
6.1. System without Turbo Coding 62
6.2. System with Turbo Coding 63

CHAPTER 7. CONCLUSION AND FUTURE WORK 65

BIBLIOGRAPHY 66
LIST OF FIGURES

1.1 Block diagram of ideal SDR. 2
1.2 Block diagram of realizable SDR. 4
2.1 Example for turbo encoder. 6
2.2 Block diagram of turbo decoder. 8
2.3 Performance of turbo code with $r = \frac{1}{2}$, $k = 5$, $L = 65536$ for various number of decoder iterations. 13
2.4 Selective concatenation scheme. 15
2.5 Performance of selective concatenation scheme. 15
3.1 Block diagram of GNU radio components. 16
3.2 Block diagram of realizable SDR using GNU radio and USRP. 17
3.3 Picture of USRP motherboard. 18
3.4 Picture of USRP with four daughter boards. 18
3.5 USRP block diagram. 19
3.6 RFX400 daughter board. 22
3.7 VERT400 antenna. 22
3.8 Block diagram of digital down converter [10]. 23
3.9 Block diagram of digital up converter [10]. 24
3.10 USRP transmit and receive signal paths [10]. 25
3.11 Flow graph of dial-tone generator. 32

4.1 Constellation diagram of BPSK. 34
4.2 Performance of different PSK schemes. 35
4.3 System model for BPSK modulation. 35
4.4 Complete model for transmission of data using BPSK. 41
4.5 Flow chart for transmission of data using BPSK. 42
4.6 System model for BPSK demodulation. 43
4.7 Complete model for transmission of data using BPSK. 51
4.8 Flow chart for transmission of data using BPSK. 52
4.9 Picture of transmitter sending packets. 54
4.10 Picture of receiver receiving packets. 54

5.1 Block diagram of transmitter using turbo coding. 57
5.2 Block diagram of receiver using turbo coding. 58
5.3 Picture of transmitter sending packets with turbo coding. 59
5.4 Picture of receiver receiving packets with turbo coding. 60

6.1 Block diagram of transmitter to transmit audio file. 63
6.2 Block diagram of receiver to receive audio file. 63
6.3 Block diagram of transmitter to transmit audio file using turbo coding. 64
6.4 Block diagram of receiver to receive audio file using turbo coding. 64
1.1. Channel Coding

Channel coding is a technique to ensure that the transmitted signal is recovered at the destination with very low probability of error. The error detection and correction is achieved through adding redundant bits to the transmitted bits. Each redundant bit is obtained from the information bits based on a predetermined rule. The output after channel coding may or may not contain the original information. The codes which include the original information at the output are called systematic codes, while those that do not include the original information are called non systematic codes.

The two main categories of channel codes are [20]:

(i) Block codes:
A block of $k$ message bits is encoded to give a codeword of $n$ bits ($n > k$). There is a unique codeword of $n$ bits for each sequence of $k$ message bits. Hamming codes and cyclic codes are examples of block codes.

(ii) Convolutional codes:
The coded sequence depends on the present and the previous information bits. It works on bit streams of arbitrary length.

Block codes and convolutional codes can be combined to further increase the performance of the communication system. These concatenated codes are used in satellite and deep space communications [17].

Turbo coding is an iterative soft-decoding algorithm that combines two or more convolutional codes and an interleaver to produce a block code that can approach Shannon limit.
Turbo codes perform better than the convolutional codes but the performance is limited by the decoder latency. More about turbo coding is discussed in chapter 2.

1.2. Software Defined Radio (SDR)

A software defined radio (SDR) system is a communication system in which the hardware components are implemented in terms of software on a personal computer. The block diagram of an ideal SDR is shown in Figure 1.1. The idea behind SDR is to implement the modulation and demodulation with software code rather than using a dedicated circuitry. The benefit of using SDR is to handle different types of signals just by loading an appropriate program rather than building circuitry for each of them. Therefore, a single SDR system can be used as one of many radio frequency (RF) transceivers (e.g., global positioning system (GPS), 802.11, high-definition television (HDTV)) by executing a different block of code.

![Figure 1.1. Block diagram of ideal SDR.](image)

In reality, the ideal SDR in Figure 1.1 is not realizable because:
• Antennas are intended to operate within a particular range of frequencies and they cannot operate on entire frequency range.

• High speed analog-to-digital converters (ADC) and digital-to-analog converters (DAC) that are used currently are not fast enough to process a large portion of occupied spectrum, and

• General purpose computers that are used currently are still inadequate to handle some real-time applications.

1.3. Realizable SDR

Currently SDRs are implemented by making use of field programmable gate arrays (FPGA) and super heterodyne mixing stages called RF front ends as shown in Figure 1.2. RF front ends are used to translate the signal from its carrier frequency to an intermediate frequency (IF) or vice versa. This helps in reducing the data rates of ADC/DACs since the ADC/DAC needs to convert the signal over its modulation bandwidth rather than using entire bandwidth. FPGAs can be stationed between the ADC/DAC and the computer in order to reduce the computational burden of the computer by performing computationally expensive signal processing techniques, like digital down/up conversions and decimation/interpolation filtering on, FPGA. The new SDR system is limited to operate in a particular frequency band (due to RF front ends) without any significant affect to flexibility.

In this thesis, I implement the turbo coding over SDR. Turbo codes are a class of iterative channel codes which demonstrates the strong capability for error correction. The advantage of SDR is that I can implement different modulation schemes and tune to any frequency band by means of software that can control the programmable hardware. So in this thesis, I am implementing a communication system which makes use of advantages of both turbo codes and SDR. In this report, chapter 3 introduces GNU radio and also explains the installation and usage of GNU radio. Chapter 4 deals with the implementation of binary phase shift keying.
Figure 1.2. Block diagram of realizable SDR.

(BPSK) modulation and demodulation. Chapter 5 deals with the implementation of turbo codes and the performance of both systems is compared in this chapter.
CHAPTER 2

TURBO CODES

2.1. Introduction

Turbo codes are high-performance forward error correcting codes (FEC) with practical decoding algorithm that closely approach the channel capacity proposed by Shannon. In the presence of noise, if one wants to achieve reliable communication with constraints in latency turbo codes are used.

Shannon’s equation:

Shannon’s equation specifies the maximum rate at which we can transmit information based on the bandwidth, the signal level and the noise level as shown below:

\[ C = B \log_2(1 + \frac{S}{N}) \]

where

- \( C \) indicates channel capacity (bits per second)
- \( B \) indicates the bandwidth of the channel (Hz)
- \( S \) indicates the total received signal power (watts)
- \( N \) indicates the total noise (watts)

2.2. Turbo Encoding

A turbo encoder is designed by parallel concatenation of two recursive systematic convolutional (RSC) codes separated by an inter leaver. Figure 2.1 shows an example of turbo encoder. Here, the two encoders are \( \frac{1}{2} \) rate RSC encoders. The upper encoder receives the data directly where as the lower encoder receives the data after it has been interleaved by a
permutation function $\alpha$. The interleaver $\alpha$ is in general a pseudo random interleaver, i.e. it maps bits in position $i$ to position $\alpha(i)$ according to a prescribed but randomly generated rule. The interleaver operates in a block wise fashion, interleaving $L$ bits at a time. Since both encoders are systematic, only one of the systematic data needs to be sent. The systematic data of the top encoder is transmitted while that of bottom encoder is not transmitted. The overall code rate of a turbo code composed of two rate $\frac{1}{2}$ RSC encoders is $\frac{1}{3}$. This code rate can be increased to $\frac{1}{2}$ by puncturing. The code rate of turbo codes can be increased to $\frac{1}{2}$ by transmitting odd indexed parity bits from upper encoder and even indexed parity bits from lower encoder [1].

![Example for turbo encoder.](image)
2.2.1. Recursive Encoding

In coded systems, performance is dominated by low weight code words. In a good code, low weight outputs are produced with very low probability. An RSC code produces low weight outputs with fairly low probability. The probability that the encoders will have inputs that produce low weight outputs is very low due to presence of the interleave. Therefore the combination of these encoders will produce a good code.

2.2.2. Pseudo Random Interleaving

In order to achieve the channel capacity proposed by Shannon, we need to develop a random code with large block length. However, decoding the random codes is extremely complex relatively sometimes it may not be possible to decode. Therefore codes must have structure in order to decode them with reasonable complexity. But codes with fixed structure don’t perform like random codes. To solve this problem we need to design a code that appears random but with structure to permit decoding. The pseudo-random interleave solves this problem.

2.3. Turbo Decoding

An estimate of the information sent can be found by solving a posteriori log-likelihood ratios given by the equations (for further information refer to the reference [1]):

\[
\Lambda_i^{(1)} = \log \frac{P[m_i = 1|y^{(0)}, y^{(1)}, z^{(2)}]}{P[m_i = 0|y^{(0)}, y^{(1)}, z^{(2)}]} \quad (2.3.1)
\]

\[
\bar{\Lambda}_i^{(2)} = \log \frac{P[\tilde{m}_i = 1|\tilde{y}^{(0)}, y^{(2)}, \tilde{z}^{(1)}]}{P[\tilde{m}_i = 0|\tilde{y}^{(0)}, y^{(2)}, \tilde{z}^{(1)}]} \quad (2.3.2)
\]

where

\(y^{(0)}\) is the observed information bits,

\(y^{(1)}\) is the observed parity bits from encoder on the top,

\(y^{(2)}\) is the observed parity bits from encoder on the bottom,

\(\tilde{y}^{(0)}\) is the interleaved version of \(y^{(0)}\)
Λ is the a posteriori log-likelihood ratio (LLR)

z is the extrinsic information which is related to LLR by

\[ z_i^{(1)} = \Lambda_i^{(1)} - y_i^{(0)} - z_i^{(2)} \]  \hspace{1cm} (2.3.3)

\[ z_i^{(2)} = \tilde{\Lambda}_i^{(2)} - \tilde{y}_i^{(0)} - \tilde{z}_i^{(1)} \]  \hspace{1cm} (2.3.4)

In order to solve the equations from (2.3.1) to (2.3.4), we make use of the structure shown in Figure 2.2. Decoder1 determines solution to equation (2.3.1) and decoder2 determines solution to equation (2.3.2). Each decoder calculates the LLR of the information it receives and sends the extrinsic information to the other decoder and this process is repeated for certain number of iterations. The final estimate of data is obtained by hard limiting the output of one of the decoders [1].

Figure 2.2. Block diagram of turbo decoder.
\[ m_i = \begin{cases} 1 & \text{if } \Lambda^{(2)}_i > 0 \\ 0 & \text{if } \Lambda^{(2)}_i < 0 \end{cases} \] (2.3.5)

The a posteriori LLR’s of (2.3.1) and (2.3.2) are computed by making use of Max-Log-MAP algorithm.

### 2.3.1. MAP Algorithm

The maximum a posteriori (MAP) decoding algorithm is a technique that computes LLR of each bit recursively based on the entire observed data block of length \( N \) (for further information refer to the reference [2]).

\[ \Lambda_1(d_k) = \log \frac{Pr(d_k = 1|R_1^N)}{Pr(d_k = 0|R_1^N)} \] (2.3.6)

\( Pr(d_k = 1|R_1^N) \) is the a posteriori probability (APP) of the information at time \( k \), represented by \( d_k \), is equal to 1 given the entire received data. The received data sequence is represented by \( R_1^N = \{R_1, \ldots, R_k, \ldots, R_N\} \) where \( R_k = \{d'_k, y'^{n}_k\} \). The state of the encoder \( S_k \) is represented by

\[ S_k = (a_k, a_{k-1}, \ldots, a_{k-v+1}) \]

where \( a_k \) is the output of the first shift register in the RSC encoder. The conditional joint probability \( \Gamma^j_k(s) \) is defined as

\[ \Gamma^j_k(s) = Pr(d_k = j, S_k|R_1^N) \] (2.3.8)

The APP of \( d_k \) is equivalent to

\[ Pr(d_k = j|R_1^N) = \sum_s \Gamma^j_k(s), \quad j=0,1 \] (2.3.9)
Therefore the LLR in equation (2.3.6) changes to

\[ \Lambda_1(d_k) = \log \frac{\sum_s \Gamma_1^1(s)}{\sum_s \Gamma_2^2(s)} \] (2.3.10)

If the system is at state \( s' \) at time \( k-1 \) then equation (2.3.10) can be written in terms of joint probabilities as

\[ \Lambda_1(d_k) = \log \frac{\sum_s \sum_s' \Pr(d_k = 1, S_k = s, S_{k-1} = s', R_1^N)}{\sum_s \sum_s' \Pr(d_k = 0, S_k = s, S_{k-1} = s', R_1^N)} \] (2.3.11)

The joint probabilities in the above equation can be defined in terms of three parameters given below

\[ \alpha_k(s) = \Pr(S_k = s|R_1^k) \] (2.3.12)

\[ \beta_k(s) = \frac{\Pr(R_{k+1}^N|S_k = s)}{\Pr(R_{k+1}^N|R_1^k)} \] (2.3.13)

\[ \gamma_j(R_k, s', s) = \Pr(d_k = j, S_k = s, R_k|S_{k-1} = s') \] (2.3.14)

The LLR in equation (2.3.11) can expressed as

\[ \Lambda_1(d_k) = \log \frac{\sum_s \sum_s' \gamma_j(R_k, s', s) \alpha_{k-1}(s') \beta_k(s)}{\sum_s \sum_s' \gamma_0(R_k, s', s) \alpha_{k-1}(s') \beta_k(s)} \] (2.3.15)

\( \alpha_k(s) \) and \( \beta_k(s) \) are computed by forward and backward recursions respectively based on \( \gamma_j(R_k, s', s) \) as shown below

\[ \alpha_k(s) = h_\alpha \sum_{s'} \sum_j \gamma_j(R_k, s', s) \alpha_{k-1}(s') \] (2.3.16)

\[ \beta_k(s) = h_\beta \sum_{s'} \sum_j \gamma_j(R_{k+1}, s, s') \alpha_{k+1}(s') \] (2.3.17)

where \( h_\alpha \) and \( h_\beta \) are the normalization factors.

The branch transition probability \( \gamma_j(R_k, s', s) \) is given by

\[ \gamma_j(R_k, s', s) = \Pr(R_k|d_k = j, S_k = s, S_{k-1} = s') \times \Pr(d_k = j|S_k = s, S_{k-1} = s') \times \Pr(S_k = s|S_{k-1} = s') \] (2.3.18)
The first term in the R.H.S of equation (2.3.18) is the transition probability of the discrete channel; the second term is equal to 0 or 1 depending on whether \( d_k = j \) when the system transitions from state \( s' \) to state \( s \); and the third term is the state transition probability. The first term is divided into two terms as shown below

\[
Pr(R_k|d_k = j, S_k = s, S_{k-1} = s') = Pr(d'_k|d_k = j, S_k = s, S_{k-1} = s') \times Pr(y'_r|d_k = j, S_k = s, S_{k-1} = s')
\]  

(2.3.19)

The steps involved in MAP algorithm are given below:

(i) Initialize \( \alpha_0(S_0) \) and \( \beta_N(S_N) \) as follows:

\[
\alpha_0(S_0) = \begin{cases} 
1 & \text{for } S_0 = 0 \\
0 & \text{otherwise}
\end{cases}
\]

\[
\beta_N(S_N) = \begin{cases} 
1 & \text{for } S_N = 0 \\
0 & \text{otherwise}
\end{cases}
\]

(ii) The decoder computes \( \gamma_j(R_k, s', s) \) for \( j = 0 \) and \( 1 \) upon receiving each \( d'_k \) and its corresponding \( y'_r \). After that it computes and stores the values of \( \alpha_k(s) \) for all \( s \) based on equation (2.3.16).

(iii) After collecting all the data the backward recursion for \( \beta_k(s) \) is performed based on equation (2.3.17).

(iv) After that LLR is computed based on equation (2.3.15).

2.3.2. Max-Log-MAP Algorithm

MAP algorithm is computationally very intensive as well as too complex for real-time implementation. To avoid the complicated operations, we take logarithm of the terms in equations (2.3.12)-(2.3.14) [2].
By taking natural logarithm on both sides of equation (2.3.18), we have

\[ \tilde{\gamma}_j(R_k, s', s) = ln \gamma_j(R_k, s', s) = ln(Pr(R_k | d_k = j, S_k = s, S_{k-1} = s')) + ln(Pr(d_k = j | S_k = s, S_{k-1} = s')) + \ln(Pr(S_k = s | S_{k-1} = s')) \]

The first term on the right side of the above equation is an exponent when we consider the additive white Gaussian noise (AWGN) environment. Therefore by taking natural logarithm we can eliminate the exponent.

By taking the natural logarithm of the forward recursion \( \tilde{\alpha}_k(s) \), we have

\[ \tilde{\alpha}_k(s) = \ln(\alpha_k(s)) = h_\alpha \sum_{s'} \sum_{j=0}^{1} \gamma_j(R_k, s', s) \alpha_{k-1}(s') \]

An approximation to the logarithm of sum of number is the logarithm of the maximum number, i.e.

\[ \ln(A + B + C + ....) \approx \ln(\max(A, B, C, ....)) \]

Therefore, \( \tilde{\alpha}_k(s) \) is approximated as

\[ \tilde{\alpha}_k(s) = \ln(\alpha_k(s)) \approx \max(\tilde{\gamma}_j(R_k, s', s) + \tilde{\alpha}_{k-1}(s') + \ln h_\alpha) \]

Similarly \( \tilde{\beta}_k(s) \) can be approximated as

\[ \tilde{\beta}_k(s) = \ln(\beta_k(s)) \approx \max(\tilde{\gamma}_j(R_{k+1}, s, s') + \tilde{\beta}_{k+1}(s') + \ln h_\beta) \]

Thus LLR is computed as

\[ \tilde{\Lambda}_1(d_k) \approx \max(\tilde{\gamma}_1(R_k, s', s) + \tilde{\alpha}_{k-1}(s') + \tilde{\beta}(s)) - \max(\tilde{\gamma}_0(R_k, s', s) + \tilde{\alpha}_{k-1}(s') + \tilde{\beta}(s)) \]

For further information refer to [4] [5] [6].
2.4. Simulated Results

A turbo code with $r = \frac{1}{2}$, $k = 5$ and $L = 65536$ is chosen. The bit error rate for different values of signal-to-noise ratio (SNR) for various number of decoder iterations is calculated and plotted using MATLAB. The code for turbo encoding and decoding is written in C programming language and interfaced it with the MATLAB to obtain the simulation results shown in Figure 2.3. From the figure, we can see that as the number of decoder iterations increases the system performance increases.

![Figure 2.3. Performance of turbo code with $r = \frac{1}{2}$, $k = 5$, $L = 65536$ for various number of decoder iterations.](image)

2.5. Performance Factors

There are many factors that affect the performance of turbo codes. The most influential parameter is the interleaver size. As the interleaver size increases, performance improves.
However, as the interleaver size increases decoder latency increases. Therefore there must be a compromise between the interleaver size and the decoder latency.

While turbo codes offer extraordinary performance for bit error rates down to $10^{-5}$, the performance for very small error rates is not impressive. Bit error floor appear for low bit error rates. In order to reduce the error floors that appear at high signal-to-noise ratio selective serial concatenation can be used along with the turbo coding.

2.6. Selective Serial Concatenation of Turbo Codes

At high SNRs, only a small number of bits at the output of the turbo decoder are likely to be in error. To protect these error prone bits BCH codes can be used. Therefore by serially concatenating the Bose and Ray-Chaudhuri (BCH) code and the turbo code the error floor can be reduced. This serial concatenation technique offers higher throughput efficiency and lower complexity compared to others.

The selective concatenation scheme is shown in Figure 2.4. The $k$ most error prone bits are first identified and then encoded using a $(n, k)$ systematic BCH outer code. The $n-k$ parity bits and a 32-b cyclic redundancy check (CRC) sum are appended to the data sequence. It should be noted that the CRC only serves to stop the iterations in the turbo decoder. The length $N$ data block is then encoded using a turbo inner code whose output is modulated and transmitted over the channel. At the receiver, the received signal is demodulated and decoded using a turbo decoder. If the turbo decoder fails the CRC after $M$ iterations, the BCH decoder is invoked to decode the $k$ most error prone output bits from the turbo decoder [3].

2.6.1. Results

The above selective serial concatenation scheme was simulated using a rate $\frac{1}{3}$ turbo code with generator polynomial $(37, 33)_8$ and a block length of 2048 is used. A $(63, 51)$ double error correcting systematic BCH code was used as the outer code. Figure 2.5 compares the
BER for the proposed system with a turbo-coded system without serial concatenation. From the figure, we can conclude that the proposed serial concatenation scheme performs better than the system with just turbo coding.

Figure 2.4. Selective concatenation scheme.

Figure 2.5. Performance of selective concatenation scheme.
3.1. Introduction

GNU radio is an open source software package consisting of various signal processing blocks and an interface to tie these blocks in order to build various SDRs. To build a radio by using GNU radio, we need to create a graph in which the nodes represent the signal processing blocks and the edges represent the data flow between the processing blocks. Figure 3.1 shows the connections between the GNU radio components. C++ programming language is used to implement the various signal processing blocks and python language is used to construct and run the graphs. A signal processing block has attributes like the number of input and output ports and the type of data that must be processed by the block. Some blocks have only output ports (sources) or input ports (sinks).

![Block diagram of GNU radio components.](image)
3.2. USRP Board

Universal software radio peripheral (USRP) board is the hardware component used to build a SDR. This board was developed by Matt Wholly for GNU radio users. ADC/DAC converters, RF front end and field programmable gate array (FPGA) are incorporated in the USRP board. FPGA is incorporated into USRP board to perform computationally expensive pre-processing techniques on the input signal. The low-cost and high speed features of the USRP board lead the GNU radio users to use those boards to implement real-time applications. USRP board has one mother board and it can accommodate up to four daughter boards. Figure 3.2 shows a realizable SDR using GNU radio and USRP. It consists of 2 Tx daughter board interfaces A(J667) and B(J669), 2 Rx daughter board interfaces A(J666) and B(J668), FPGA, ADC/DAC chips and USB interface chip. Figure 3.3 shows a picture of USRP board and Figure 3.4 shows USRP with four daughter boards connected to it. It also contains DC power input and the USB 2.0 port. Figure 3.5 shows a typical setup of the USRP board [7].

![Block diagram of realizable SDR using GNU radio and USRP.](image)

3.2.1. ADC/DAC Converters

USRP board contains 4 high-speed 12-bit ADCs at the receiving path on the mother board. The sampling rate used is 64M samples per second. In principle, we can digitize a band-pass signal of frequency up to 32MHz based on Nyquist principle. Full range on ADCs is
2V peak to peak and the input is 50ohms differential, i.e. 40mW or 16dBm. Programmable gain amplifier (PGA) is available before the ADCs for the purpose of amplifying the signal if we receive a weak signal. The PGA can provide a gain up to 20dB. The available sampling rates are 64 MS/s, 42.66 MS/s, 32 MS/s, 25.6 MS/s and 21.33 MS/s (S indicates samples).
USRP board also has 4 high-speed 14-bit DACs at the transmitting path on the mother board. The DAC clock frequency is 128 samples per second. The DACs can supply a peak of 1V to a 50ohm differential load, i.e. 10mW or 10dBm. PGA is also available after the DAC to provide a gain up to 20dB [7].

3.2.2. Daughter Boards

We can plug in up to 2 Rx daughter boards and 2 Tx daughter boards on the mother board of USRP board. The slots for 2 Tx daughter boards are labeled TxA and TxB, and the corresponding Rx daughter board slots are labeled as RxA and RxB. Each Tx daughter board slot has access to 2 high-speed DACs and each Rx daughter board has access to 2 high-speed ADCs as shown in Figure 3.5.

Each Tx daughter board has 2 differential analog outputs which are updated at 128 MS/s. Each Rx daughter board has a pair of differential analog inputs which are sampled at a rate of 64 MS/s. An I2C electrically erasable programmable read-only memory (EEPROM) is provided in each daughter board to send the information of the board to the system. Based on the information in EEPROM the host software can automatically setup the system. The various kinds of daughter boards available are [8]:

Figure 3.5. USRP block diagram.
(i) Basic Tx and Basic Rx:

Frequency range: 1MHz to 250MHz

They are designed for use with external RF front ends.

(ii) LFTX and LFRX:

Frequency range: DC to 30MHz

They are similar to Basic Tx and Basic Rx. Their frequency response extends down to DC because of use of differential amplifiers instead of transformers. They also have 30MHz low pass filters for anti-aliasing.

(iii) TVRX:

Frequency range: 50MHz - 860 MHz

The TVRX daughter board is a complete VHF and UHF receiver system based on a TV tuner module. By simply connecting an antenna we can receive a 6MHz wide block of spectrum in the 50MHZ - 860MHZ range.

(iv) DBSRX:

Frequency range: 800MHz - 2.4GHz

The DBSRX is a complete receiver system for 800MHz to 2.4GHz with a 3-5dB noise figure.

We can connect 2 Transceiver daughter boards on the mother board by making use of four slots that are available. The various kinds of Transceiver daughter boards available are [9]:

(i) WBX:

Frequency range: 50MHz - 2.2GHz

Transmit power: 100mW (20dBm)

(ii) RFX400:

Frequency range: 400MHz - 500MHz

Transmit power: 100mW (20dBm)
(iii) RFX900:
Frequency range: 750MHz - 1050MHz
Transmit power: 200mW (23dBm)

(iv) RFX1200:
Frequency range: 1150MHz - 1450MHz
Transmit power: 200mW (23dBm)

(v) RFX1800:
Frequency range: 1.5GHz - 2.1GHz
Transmit power: 100mW (20dBm)

(vi) RFX2400:
Frequency range: 2.3GHz - 2.9GHz
Transmit power: 50mW (17dBm)

(vii) XCVR2450:
Frequency range: 2.4GHz - 2.5GHz and 4.9GHz - 5.9GHz
Transmit power: 100mW (20dBm)

I am using two USRPs each equipped with a transceiver RFX400 daughter board as shown in Figure 3.6. The antenna which I am using is VERT400. It is a 7-inch vertical antenna which can work with RFX400 daughter board. Figure 3.7 shows a picture of VERT400 antenna.

3.2.3. FPGA

Understanding the purpose of FPGA on the motherboard of USRP is essential for GNU radio users. All the ADCs and DACs are connected to the FPGA. The role of FPGA is to perform computationally expensive math and to reduce the data rates so that we can transfer the data using USB2.0. The FPGA is connected to Cypress FX2 which provides an interface to USB2. The USB2 bus is used to program everything on the USRP. The FPGA used in
USRP is Altera Cyclone EP1C12Q240C8.

FPGA is configured to perform operations like digital up-conversion, digital down-conversion, decimation and interpolation. Digital down converters (DDC) are implemented by making use of cascaded integrator-comb (CIC) filters. CIC filters are very high performance
filters designed using only adds and delays. Four digital down converters are implemented in FPGA allowing 1, 2 or 4 separate Rx channels. Each digital down converter has two inputs I (in-phase) and Q (quadrature phase). Each of the four ADCs can be routed to either of I or Q input of any of the 4 DDCs. This allows implementing multiple channels with same ADC rate in each channel. Figure 3.8 shows DDC and decimation stage in the USRP receive path.

![Block diagram of digital down converter](image)

Figure 3.8. Block diagram of digital down converter [10].

The digital up converters (DUCs) are present on the transmit side of the USRP. The DUCs are contained in the AD9862 CODEC chips but not in FPGA. On the transmit side of the USRP only interpolators are implemented in FPGA. The output of the interpolator can be routed to any one of the four CODEC inputs. Figure 3.9 shows the DUC in the USRP transmit path [10].

3.2.4. USRP Transmit and Receive paths

Receive Signal Path:

Figure 3.10 shows the USRP transmit and receive signal paths. The USRP accommodates
two receiver daughter boards. The slots available on USRP to connect two receiver daughter boards are labeled RxA and RxB and the interfaces connecting these daughter boards are labeled J666 and J668 respectively. Two real-valued voltage signals, labeled VIN_A_X and VIN_B_X (where X is replaced by A or B based on the receiver slot to which the signal belongs), from the daughter board are fed to the interface [10].

The four analog signals VIN_A_A, VIN_A_B, VIN_B_A and VIN_B_B are sent to four different ADCs. Each 12-bit ADC samples the signals at 64MS/s and sends them to FGPA for processing. The digitized signals are routed by a MUX to the appropriate DDC.

The DDC accepts two inputs, in-phase and quadrature inputs. Based on the user specifications VIN_A_A, VIN_A_B, VIN_B_A, VIN_B_B or all zeros is connected to the in-phase or quadrature input of each of the four DDCs. Each DDC mixes the baseband signal to its input signal. Upon mixing with the baseband, the signal is decimated by a user specified factor. Based on the factor specified by the user (assume M), the signal is first decimated by a factor of M/2 using CIC filters and then decimated by 2 using half-band filter. Figure 2.4 shows the DDC and the decimation stage. The output of the half-band filter is then interleaved.
Figure 3.10. USRP transmit and receive signal paths [10].

and pushed into a first-in-first-out (FIFO) buffer. These data samples are sent to the host computer for processing by the USB2.0 interface chip.

Transmit Signal Path:

The information from the host computer is pushed into the transmit FIFO. This data is sent to the interpolation stage after de-interleaving. If the user specifies an interpolation factor L then the data is interpolated by a factor of L/4 using CIC filters during interpolation stage. The output of the interpolation stage is fed to DEMUX which sends the in-phase and quadrature output of each CIC interpolator to the in-phase and quadrature inputs of one of the DACs correspondingly. The user must specify the connections between the DAC chip
and the interpolator. The signal is again interpolated by a factor of 4 inside the DACs by making use of half-band filter interpolators. The output of the half-band filter interpolators is fed to a DUC. The output of DUC is converted into samples of 14 bits each and fed to individual digital-to-analog converters to convert them to analog signals at 128MS/s. These analog signals are then sent to slots TxA or TxB using the daughter interfaces J667 or J669 respectively [10].

3.3. GNU Radio Installation

This section provides information about the background applications and libraries that are needed to install the GNU radio along with the steps that are to be followed to install the GNU radio on Ubuntu system (Lucid 10.4) [11]. The packages that are required to compile various components of GNU radio are listed below:

(i) Development Tools:

- g++
- subversion
- make
- autoconf, automake, libtool
- sdcc
- guile
- ccache

(ii) Libraries:

- python-dev
- FFTW 3.X(fftw3, fftw3-dev)
- cppunit(libcppunit and libcppunit-dev)
- Boost 1.35(or later)
- libusb and libusb-dev
- wxWidgets(wx-common) and wxPython(python-wxgtk2.8)
- python-numpy
- ALSA(alsa-base, libsound2 and libsound2-dev)
- Qt(libqt3-mt-dev)
- SDL(libsdl-dev)

(iii) SWIG
(iv) QWT
(v) QWT Pot3d Lib
(vi) Other useful packages
  - doxygen
  - octave

These packages can be installed via ‘synaptic package manager’ or ‘apt-get’. We can simply install these packages by opening a new terminal and entering the below command in the terminal.

```
sudo apt-get -y install libfontconfig1-dev libxrender-dev libpulse-dev swig g++ automake libtool \ python-dev libfftw3-dev libcppunit-dev libboost-all-dev libusb-dev fort77 sdcc sdcc-libraries \ libSDL1.2-dev python-wxgtk2.8 subversion git-core guile-1.8-dev \ libqt4-dev python-numpy ccache python-opengl libgsl0-dev \ python-cheetah python-lxml doxygen qt4-dev-tools \ libqwt5-qt4-dev libqwtplot3d-qt4-dev pyqt4-dev-tools
```

3.3.1. Installing GNU Radio

To install the GNU radio package type the commands shown below inside the terminal one by one.
#Installing GNU radio from git

```bash
git clone http://gnuradio.org/git/gnuradio.git
cd gnuradio
export LD_LIBRARY_PATH=$BOOST_PREFIX/lib
./bootstrap
./configure --with-boost=$BOOST_PREFIX
make
```

Now run the GNU radio software self-check by typing the command given below and make sure that the tests doesn’t fail.

```bash
make check
```

Now install GNU radio by using the command

```bash
sudo make install
```

Now setup groups to handle USRP through USB using the commands

```bash
sudo addgroup usrp
sudo usermod -G usrp -a <YOUR_USERNAME>
echo 'ACTION=="add", BUS=="usb", SYSFSidVendor=="ffe", SYSFSidProduct=="0002", GROUP=="usrp", MODE=="0660"' > tmpfile
sudo chown root.root tmpfile
sudo mv tmpfile /etc/udev/rules.d/10-usrp.rules
```

Now reload the rules of ‘udev’ in order to configure the Ubuntu system to perform certain task when it detects the USRP on the USB by using the command

```bash
sudo udevadm control --reload-rules
```

We can check whether USRP is detected by the Ubuntu system by using the command

```bash
ls -IR /dev/bus/usb | grep usrp
```

If it returns something then the USRP is detected. Now, test whether the GNU radio radio
works with USRP or not, using an example ‘usrp_benckmark_usb’. To run this example use the commands

```
cd gnuradio-examples/python/usrp
./usrp_benchmark_usb.py
```

If the program runs properly, then GNU radio and USRP are installed and working properly. Otherwise restart the system and follow the steps shown below:

(i) Make a copy from the current ld.so.conf file and save it in a temp folder:
```
cp /etc/ld.so.conf /tmp/ld.so.conf
```

(ii) Add /usr/local/lib path to it:
```
echo /usr/local/lib >> /tmp/ld.so.conf
```

(iii) Add Boost library path to the file:
```
echo /opt/boost_1.37.0/lib >> /tmp/ld.so.conf
```

(iv) Delete the original ld.so.conf file and put the modified file instead:
```
sudo mv /tmp/ld.so.conf /etc/ld.so.conf
```

(v) Do ldconfig:
```
sudo ldconfig
```
After running these commands restart the system and run the example program once again, it must work now.

3.4. Dial-tone Program

Let’s go through an example program to understand the usage of the signal processing blocks provided by the GNU radio package in python programming. The dial tone program is often called the ‘Hello world of GNU radio’. This example program includes the generation of dial tone and playing it using audio device available on PC. The dial tone is generated by two sine waves at different frequencies, one on the right channel and the other on the left channel.
of the audio device.

```python
#!/usr/bin/env python
from gnuradio import gr
from gnuradio import audio

class my_top_block(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self)
        sampling_rate = 32000
        ampl = 0.1
        src0 = gr.sig_source_f(sampling_freq, gr.GR_SIN_WAVE, 350, ampl)
        src1 = gr.sig_source_f(sampling_freq, gr.GR_SIN_WAVE, 440, ampl)
        dst = audio.sink(sampling_f,"")
        fg.connect((src0, 0), (dst, 0))
        fg.connect((src1, 0), (dst, 1))

if __name__ == '__main__':
    try:
        my_top_block().run()
    except KeyboardInterrupt:
        pass
```

First line in the code ‘#!/usr/bin/env python’ helps us to run the program directly from the command line by giving the program file an executable mode. In order to give the executable mode to the program file, we need to execute the command ‘chmod +x filename’.

To use the signal processing blocks that are predetermined in the GNU radio package,
we need to import the modules containing those blocks from the packages of GNU radio. ‘gnuradio’ is the package which includes all the GNU related modules. Lines 2 and 3 import the modules that are needed to run the program on GNU radio from the gnuradio package using the import command. In this program, we are importing the modules gr and audio from gnuradio package. The module ‘gr’ is the basic GNU radio module that must be imported to run a GNU radio application. The module ‘audio’ is imported to load the audio device blocks.

Next we define a class called my_top_block which is derived from gr.top_block class, and contains the flow graph. The functions needed to add the blocks and to connect them are provided by the gr.top_block class. Line 5 declares the initialization method __init__. __init__ is an important method for every class. It is often referred to as a constructor. Every time we create an object of the type my_top_block, we call the __init__ method to initialize the object. The first thing the method __init__ of my_top_block class does is to call the initialization method of the base class gr.top_block(Line 6). Lines 7 and 8 defines two variables sampling_rate and ampl to control the sampling rate and the amplitude of the signals generated. Lines 9-13 describe the flow graph of the dial tone program shown in Figure 3.11. Lines 9 and 10 generate two signal sources(src0 and src1). These two sources generate continuous sine waves at frequencies 350Hz and 440Hz at a sampling rate of 32kHz. The amplitude of these sine waves is set to 0.1 using the variable ampl. The ‘f’ in gr.sig_source_f indicates that the output is of type float. Line 11 declares an audio sink(dst) which plays back the samples piped into it. The sampling rate of the audio sink is set to the sampling rate of the generated signals. Lines 12 and 13 connects these blocks as shown in Figure 3.11. self.connect(block1,block2,block3,.....) is the general syntax for connecting the blocks. By making use of this syntax, we can connect the output of block1 to the input of block2, the output of block2 to the input of block3 and so on. But in this example we need to connect the outputs of both src0 and src1 to the input of dst. Line 12 connects the output of src0 to the input port 0 of the dst and line
13 connects the output of src1 to the input port 1 of the dst. Further details are provided in [13].

![Flow graph of dial-tone generator.](image)

Figure 3.11. Flow graph of dial-tone generator.

The last 5 lines are used to run the flow graph. The try and except statements make sure that the flow graph is stopped when ctrl+z is pressed on the keyboard.
CHAPTER 4

BPSK MODULATION

4.1. Introduction

The various digital modulation techniques that can be used to transmit digital data can be grouped under three major classes as follows:

- Amplitude shift keying (ASK)
- Frequency shift keying (FSK)
- Phase shift keying (PSK)

All the modulation techniques convey the data by changing an aspect of a base signal with respect to the digital signal. In case of PSK, the phase of the base signal is changed in accordance with the digital signal. The two fundamental ways in which we can utilize the phase of the signal is:

(i) In the first case, the information is conveyed by the phase of the signal itself. In this case, the demodulator should have a reference signal to compare the phase of the received signal with it to obtain the information.

(ii) In the second case, the information is conveyed by the changes in the phase of the signal (differential schemes).

Constellation diagram is used to conveniently represent the various PSK schemes. The constellation diagram shows the points in z-plane where the real and imaginary axes are replaced by the in-phase and quadrature phase components respectively. Based on this representation we can easily implement the modulation scheme. The amplitude of each point along the in-phase axis is used to modulate a cosine (or sine) wave and the amplitude along the quadrature axis to modulate a sine (or cosine) wave.
In PSK, the constellation points are positioned around a circle with uniform angular spacing in order to obtain maximum phase separation between adjacent points to provide immunity to corruption. They can be transmitted with the same energy since they are positioned on a circle. In PSK scheme, the number of constellation points will be a power of 2 since the data to be conveyed is binary.

Binary phase shift keying (BPSK) is the simplest PSK scheme. Two phases separated by 180° are used to implement the BPSK modulation. The constellation diagram of the BPSK modulation is shown in Figure 4.1. It consists of two points on the in-phase axis, one at 0° and the other at 180°, i.e. bit 0 is represented by the point ‘-1’ on the in-phase axis and bit 1 is represented by the point ‘1’ on the in-phase axis. BPSK gives better performance when compared with other PSK schemes since the separation between the constellation points is more in case of BPSK. Figure 4.2 shows the performance of different PSK schemes. But it is not suitable for high data rate applications [19].

![Figure 4.1. Constellation diagram of BPSK.](image)

4.2. BPSK Modulation

4.2.1. System Model

Figure 4.3 shows the system model for the BPSK modulation. The data to be modulated is first sent to bytes-to-chunks converter block. In this block, the received bytes are converted
Figure 4.2. Performance of different PSK schemes.

into k bit vectors. The converted vectors are then sent to symbol mapper block. In this block, the bits are mapped to symbols, i.e. bit 0 is mapped to -1 and bit 1 to 1. The output of symbol mapper is then fed to chunks-to-symbol converter. This block maps a stream of symbols to a stream of complex constellation points. The output of this block is then fed to root raised cosine (RRC) filter. The RRC filter is used in digital communication system as transmit and receive filter to perform matched filtering. This block maps the constellation points to waveforms, here we map the points to root raised cosine waveform.

Figure 4.3. System model for BPSK modulation.

4.2.2. Implementation

First import the packages that are to be used to implement the BPSK modulation.
from gnuradio import gr, gru, modulation_utils
from math import pi, sqrt
import psk
import cmath

Next define the default values used in the implementation.

    _def_samples_per_symbol = 2
    _def_excess_bw = 0.35
    _def_gray_code = True
    _def_verbose = False
    _def_log = False

Next define and initialize a class bpsk_mod which is derived from the class gr.hier_block2 and that performs the BPSK modulation. We call the _init__ method to initialize the object of class bpsk_mod. The first thing the method _init__ of bpsk_mod class does is to call the initialization method of the base class gr.hier_block2 which specifies the input and output signature of the class bpsk_mod. The input to the class bpsk_mod is a byte stream (unsigned char) and the output is the complex modulated signal at baseband. After that we initialize the parameters that are used in BPSK modulation, using the default values. Here the parameter samples_per_symbol indicates the number of samples required for symbol($\geq$2) and it is of type integer. The parameter excess_bw indicates the excess bandwidth of RRC filter and it is of type float. The parameter gray_code tells the modulator to Gray code the bits and it is of type boolean. The parameter verbose prints the information about the modulator and it is of type boolean. The parameter log prints modulation data to files and it is of type boolean. After that check whether the parameter samples_per_symbol is $\geq$2, if false raise an error and exit from the class. After that define the number of taps needed for
designing the pulse shaping filter (RRC filter). The following statements perform these actions:

class bpsk_mod(gr.hier_block2):
    def __init__(self,
        samples_per_symbol=def_samples_per_symbol,
        excess_bw=def_excess_bw,
        gray_code=def_gray_code,
        verbose=def_verbose,
        log=def_log):
        gr.hier_block2.__init__(self, "bpsk_mod",
            gr.io_signature(1, 1, gr.sizeof_char), # Input signature
            gr.io_signature(1, 1, gr.sizeof_gr_complex)) # Output signature
        self.samples_per_symbol = samples_per_symbol
        self.excess_bw = excess_bw
        self.gray_code = gray_code
        if not isinstance(self.samples_per_symbol, int) or self.samples_per_symbol 2:
            raise TypeError, ("sps must be an integer = 2, is\%d\"%self.samples_per_symbol)  

    ntaps = 11 * self.samples_per_symbol

Next we implement the bytes-to-chunks converter block. The function of this block is to convert the received bytes into single bit vectors. In order to perform this action, we make use of the class packed_to_unpacked_bb provided by gr package. This class takes in bits_per_chunk and endianness as inputs. To convert the received bytes into single bit vectors, we specify the value of bits_per_chunk as ‘1’ and we select the endianness as gr.GR_MSB_FIRST. These actions are performed by the following statements:
# turn bytes into k-bit vectors
self.bytes2chunks = gr.packed_to_unpacked_bb(1, gr.GR_MSB_FIRST)

Next we implement the symbol mapper block. The function of this block is to convert the received bits to symbols. First, we need to specify whether gray coding must be used or not, by making use of the parameter `gray_code`. To perform gray coding, we need to use the method `binary_to_gray` provided by the class `psk`, otherwise use the method `binary_to_ungray` provided by the same class. These methods must be provided with the information about the number of symbols that are used by the modulation scheme (in this case it is 2). After that we need to map the bits to symbols by using the class `map_bb` provided by the `gr` package. These actions are performed by the following statements:

```python
if self.gray_code:
    self.symbol_mapper = gr.map_bb(psk.binary_to_gray[2])
else:
    self.symbol_mapper = gr.map_bb(psk.binary_to_ungray[2])
```

Next we implement the chunks-to-symbol converter block. The function of this block is to map a stream of symbols to a stream of complex constellation points. In order to perform this action, we make use of the class `chunks_to_symbols_bc` provided by `gr` package. This class takes in `symbol_table` as input which specifies the mapping function. The mapping function is specified in the method `constellation` provided by the `psk` class. These actions are performed by the following statements:

```python
self.chunks2symbols = gr.chunks_to_symbols_bc(psk.constellation[2])
```

Finally we implement the RRC filter block which performs pulse shaping operation. To perform this action, we need to use the function `root_raised_cosine` provided by the class
firdes in gr package. This function takes the gain, sampling_frequency, sampling_rate, alpha (excess_bandwidth) and ntaps as inputs. The following statements specify the RRC filter block:

```python
# pulse shaping filter
self.rrc_taps = gr.firdes.rootRaisedCosine(
    self.samples_per_symbol,  # gain (samples_per_symbol since we're
    self.samples_per_symbol,  # sampling rate
    1.0,  # symbol rate
    self.excess_bw,  # excess bandwidth (roll-off factor)
    ntaps)
self.rrc_filter = gr.interp_fir_filter_ccf(self.samples_per_symbol, self.rrc_taps)
```

Now connect all these blocks to create a flow graph which implements BPSK modulation using the statement

```python
self.connect(self, self.bytes2chunks, self.symbol_mapper, self.chunks2symbols, self.rrc_filter, self)
```

To print the parameters of the modulation, we need to make use of the parameter verbose as shown below:

```python
if verbose:
    self.print_verbage()

def _print_verbage(self):
    print "Modulator:"
    print "Gray code: %s" % self.gray_code
    print "RRC roll-off factor: %.2f" % self.excess_bw
```

39
To setup the logging files, which contain the modulation data, we need to make use of the parameter `log` as shown below:

```python
if log:
    self._setup_logging()

def _setup_logging(self):
    print "Modulation logging turned on."
    self.connect(self.bytes2chunks,
                 gr.file_sink(gr.sizeof_char, "tx_bytes2chunks.dat"))
    self.connect(self.symbol_mapper,
                 gr.file_sink(gr.sizeof_char, "tx_graycoder.dat"))
    self.connect(self.chunks2symbols,
                 gr.file_sink(gr.sizeof_gr_complex, "tx_chunks2symbols.dat"))
    self.connect(self.rrc_filter,
                 gr.file_sink(gr.sizeof_gr_complex, "tx_rrc_filter.dat"))
```

Next add BPSK modulation specific options to the standard parser by making use of the method `add_option` provided by `parser` class. The following statements specify the operation of adding the BPSK modulation specific options:

```python
def add_options(parser):
    parser.add_option("", "--excess-bw", type="float", default=_def.excess_bw,
                     help="set RRC excess bandwidth factor[default=%default]"
                     parser.add_option("", "--no-gray-code", dest="gray_code",
                     default=True, help="disable gray coding on modulated bits (PSK)"
add_options=staticmethod(add_options)
```
Next we add the BPSK modulation to the modulation registry of GNU radio using the statement

```
modulation_utils.add_type_1_mod('bpsk', bpsk_mod)
```

First create a python program combining these statements and save it as ‘bpsk.py’. Now we can call the BPSK modulation from another program by importing the bpsk.py program and using statement given below:

```
mod = bpsk.bpsk_mod(samples_per_symbol = 2, excess_bw = 0.35,)
```

4.2.3. Transmission of Data using BPSK

Figure 4.4 depicts a complete model for transmitting file using BPSK modulation.

![Complete model for transmission of data using BPSK](image)

Figure 4.4. Complete model for transmission of data using BPSK.

Figure 4.5 shows the flow chart for transmission of data using BPSK modulation. The values of frequency, data rate, packet size, size of data to be transmitted and the amplitude at which the signal must be transmitted should be specified at the command line while running the program. The frequency is specified based on the daughter board used. The frequency range of RFX400 daughter board is 400-500MHz. So we can specify a frequency in between 400MHz and 500MHz. Minimum data rate with which we can transmit the data using USRP is 32kbps. So we need to specify a data rate \( \geq 32\text{kbps} \). Maximum packet size with which we can transmit using USRP is 4096. So we need to specify packet size \( \leq 4096 \). We need to setup the USRP board based on these specifications. In order to setup the USRP to transmit based on these specifications, we need to import program usrp_transmit_path available in the
gnuradio examples provided by the GNU radio package.

After setting up the USRP, read a data of length specified by (packet_size-2) from the file to be transmitted and append a header to create a packet with size specified by the user (packet_size). Perform BPSK modulation on this packet and send it using the USRP board. Repeat this process until the entire file is transmitted.

Figure 4.5.  Flow chart for transmission of data using BPSK.
4.3. BPSK Demodulation

4.3.1. System Model

Figure 4.6 depicts the system model for BPSK demodulation. The data to be demodulated is first sent to a pre-scaler which scales the signal from full range to ±1. The output of pre-scaler is then fed to the automatic gain control (AGC) block. This block computes the gain based on the maximum obsolete value over a finite number of samples. The output of AGC block is connected to Costas loop which helps in tracking the carrier. The output of Costas loop is then passed through a RRC filter. The output of the RRC filter is connected to clock recovery block. This block implements Mueller and Muller (MM) discrete time error tracking synchronizer. The output is then fed to a slicer block which maps the constellation points to bits. The output is then fed to unpack block which converts a byte with $k$ relevant bits to $k$ output bytes with 1 bit in the LSB.

![System model for BPSK demodulation.](image)

4.3.2. Implementation

First import the packages that are needed to implement BPSK demodulation using the statements

```python
from gnuradio import gr, gru, modulation-utils
from math import pi, sqrt
import psk
import cmath
```

Next define the default values used in the implementation.
Next define and initialize a class `bpsk_demod` which is derived from the class `gr.hier_block2` and that performs the BPSK demodulation. We call the `__init__` method to initialize the object of class `bpsk_demod`. The first thing the method `__init__` of `bpsk_demod` class does is to call the initialization method of the base class `gr.hier_block2` which specifies the input and output signature of the class `bpsk_mod`. The input to the class `bpsk_demod` is the complex modulated signal at baseband and the output is a stream of bits packed 1 bit per byte. After that we initialize the parameters that are used in BPSK demodulation, using the default values. Here the parameter `samples_per_symbol` indicates the number of samples required for symbol (≥2) and it is of type integer. The parameter `excess_bw` indicates the excess bandwidth of RRC filter and it is of type float. The parameter `costas_alpha` indicates the gain of the Costas loop and it is of type float. The parameter `gain_omega` is used to adjust omega which determines the sampling period and it is of type float. The value of the parameter `mu` is related to the adjustment of the sampling due to the error signal and it is of type float and it must be between 0 and 1. The parameter `gain_mu` is the correction factor based on the timing difference between symbols and it is of type float. The parameter `gray_code` tells the modulator to gray code the bits and it is of type boolean. The parameter `verbose` prints the information about the demodulator and it is of type boolean. The parameter `log` prints modulation data to files and it is of type boolean. After that check whether the parameter `samples_per_symbol` is ≥2, if false raise an error and exit from the class. After that define the number of taps needed for designing the pulse shaping filter (RRC filter). The following statements perform these actions:
class bpsk_demod(gr.hier_block2):
    def __init__(self,
        samples_per_symbol=_def_samples_per_symbol,
        excess_bw=_def_excess_bw,
        costas_alpha=_def_costas_alpha,
        gain_mu=_def_gain_mu,
        mu=_def_mu,
        omega_relative_limit=_def_omega_relative_limit,
        gray_code=_def_gray_code,
        verbose=_defVerbose,
        log=_def_log):
        gr.hier_block2.__init__(self, “bpsk_demod”,
            gr.io_signature(1, 1, gr.sizeof_gr_complex), # Input signature
            gr.io_signature(1, 1, gr.sizeof_char)) # Output signature
        self._samples_per_symbol = samples_per_symbol
        self._excess_bw = excess_bw
        self._costas_alpha = costas_alpha
        self._gain_mu = gain_mu
        self._mu = mu
        self._omega_relative_limit = omega_relative_limit
        self._gray_code = gray_code
        if samples_per_symbol < 2:
            raise TypeError, “samples_per_symbol must be >= 2, is %r”
            % (samples_per_symbol,)

Next we implement the Pre-scaler block. The function of this block is to scale the signal
from full range to ±1. It can be done by multiplying the input with some constant. The class `multiply_const_cc` provided by `gr` package is used to multiply the input with a constant specified by the variable `scale` and it will be performed in complex domain. The following statements specify these operations:

```python
# scale the signal from full-range to +-1
scale = (1.0/16384.0)
self.pre_scaler = gr.multiply_const_cc(scale)
```

Next we implement the AGC block. The function of this block is to calculate the gain which can be performed by making use of the class `feedforward_agc_cc` available in `gr` package. It takes the number of samples and the reference float value as inputs and calculates the gain. The following statement specify the operation of AGC block:

```python
# Automatic gain control
self.agc = gr.feedforward_agc_cc(16, 2.0)
```

Next we implement the Costas loop block. The function of this block is to tracks the carrier. To perform this operation, we can use the class `costas_loop_cc` provided by the `gr` package. It takes the values of `costas_alpha`, `beta`, `max_freq`, `min_freq` and `order` as inputs to track the carrier. The value of parameter `beta` is calculated from the value of `costas_alpha`. The following statements perform the operation of Costas loop block:

```python
# Costas loop (carrier tracking)
costas_order = 4
beta = .25 * self._costas_alpha * self._costas_alpha
self.costas_loop = gr.costas_loop_cc(self._costas_alpha, beta,0.1, -0.1, costas_order)
```

Next we implement the RRC filter block which acts as a matched filter. To perform
this action, we need to use the function `root_raised_cosine` provided by the class `firdes` in `gr` package. This function takes the `gain`, `sampling_frequency`, `sampling_rate`, `alpha` (excess_bandwidth) and `ntaps` as inputs. The following statements specify the RRC filter block:

```python
# pulse shaping filter
ntaps = 11 * samples_per_symbol
self.rrc_taps = gr.firdes.root_raised_cosine(
    self.samples_per_symbol, # gain
    self.samples_per_symbol, # sampling rate
    1.0, # symbol rate
    self.excess_bw, # excess bandwidth (roll-off factor)
    ntaps)
self.rrc_filter = gr.interp_fir_filter_ccf(self.samples_per_symbol, self.rrc_taps)
```

Next we implement the symbol clock recovery block. We implement this block by making use of Mueller and Muller (MM) discrete time error tracking synchronizer which is described in the class `clock_recovery_mm_cc` provided by `gr` package. It takes the values of `omega`, `gain_omega`, `mu`, `gain_mu` and `omega_relative_limit` to perform the specified operation of error tracking. The following statements specify the operations of clock recovery block:

```python
# symbol clock recovery
omega = self.samples_per_symbol
gain_omega = .25 * self.gain_mu * self.gain_mu
self.clock_recovery = gr.clock_recovery_mm_cc(omega, gain_omega, 
    self.mu, self.gain_mu, 
    self.omega_relative_limit)
```

Next we implement the slicer block. The function of this block is to map symbols to
bits. To perform this operation we make use of classes `constellation_decoder_cb` and `map_bb` provided by `gr` package. The following statements specify the operation of slicer block:

```python
# find closest constellation point
rot = 1
rotated_const = map(lambda pt: pt * rot, psk.constellation[2])
self.slicer = gr.constellation_decoder_cb(rotated_const, range(2))
if self._gray_code:
    self.symbol_mapper = gr.map_bb(psk.gray_to_binary[2])
else:
    self.symbol_mapper = gr.map_bb(psk.ungray_to_binary[2])
```

Next we implement the unpack block. The function of this block is to unpack the $k$ bit vector into a stream of bits. For this operation we use the class `unpack_k_bits_bb` provided by `gr` package. The following statements specify the operation of unpack block:

```python
# unpack the k bit vector into a stream of bits
self.unpack = gr.unpack_k_bits_bb(1)
```

Now connect all these blocks to create a flow graph which implements BPSK demodulation using the statement

```python
# Connect and Initialize base class
self.connect(self, self.pre_scaler, self.agc, self.costas_loop, self.rrc_filter, self.clock_recovery, self.slicer, self.unpack, self)
```

To print the parameters of the demodulation, we need to make use of the parameter `verbose` as shown below:
if verbose:
    self._print_verbage()

def _print_verbage(self):
    print "Demodulator:"
    print "bits per symbol: %d" % self.bits_per_symbol()
    print "Gray code: %s" % self._gray_code
    print "RRC roll-off factor: %.2f" % self._excess_bw
    print "Costas Loop alpha: %.2e" % self._costas_alpha
    print "Costas Loop beta: %.2e" % self._costas_beta
    print "MM mu: %.2f" % self._mm_mu
    print "MM mu gain: %.2e" % self._mm_gain_mu
    print "MM omega: %.2f" % self._mm_omega
    print "MM omega gain: %.2e" % self._mm_gain_omega
    print "MM omega limit: %.2f" % self._mm_omega_relative_limit

To setup the logging files, which contain the demodulated data, we need to make use of the parameter log as shown below:

    if log:
        self._setup_logging()

def _setup_logging(self):
    print "Modulation logging turned on."
    self.connect(self.pre_scaler,
                 gr.file_sink(gr.sizeof_gr_complex, "rx_prescaler.dat"))
    self.connect(self.agc,
                 gr.file_sink(gr.sizeof_gr_complex, "rx_agc.dat"))
Next add BPSK demodulation specific options to the standard parser by making use of the method `add_option` provided by `parser` class. The following statements specify the operation of adding the BPSK demodulation specific options:

```python
def add_options(parser):
    parser.add_option('', '--excess-bw', type='float', default=_def_excess_bw,
                       help="set RRC excess bandwith factor[default=%default] (PSK)"")
    parser.add_option('', '--no-gray-code', dest='gray_code', action='store_false',
                       default=_def_gray_code, help="disable gray coding on modulated bits (PSK)"")
    parser.add_option('', '--costas-alpha', type='float', default=None,
                       help="set Costas loop alpha value[default=%default] (PSK)"")
    parser.add_option('', '--gain-mu', type='float', default=_def_gain_mu,
                       help="set MM symbol sync loop gain mu value [default=%default] (GMSK/PSK)"")
    parser.add_option('', '--mu', type='float', default=_def_mu,
                       help="set MM symbol sync loop mu value[default=%default] (GMSK/PSK)"")
```

50
parser.add_option("", "–omega-relative-limit", type="float",
    default=_def_omega_relative_limit,
    help="MM clock recovery omega relative limit [default=%default] (GMSK/PSK)"
    add_options=staticmethod(add_options)

    Next we add the BPSK demodulation to the demodulation registry of GNU radio using
    the statement

    modulation_utils.add_type_1_demod(‘bpsk’ , bpsk_demod)

    First create a python program combining these statements and save it into ‘bpsk.py’. Now we can call the BPSK demodulation from another program by importing the bpsk.py program and using statement given below

    demod = bpsk.bpsk_demod(samples_per_symbol = 2, excess_bw = 0.35, costas_alpha = 0.175,)

    For further information on the classes refer [18].

4.3.3. Receiver

    Figure 4.7 depicts a complete model for receiving a file using BPSK demodulation.

![Figure 4.7. Complete model for transmission of data using BPSK.](image)

    Figure 4.7. Complete model for transmission of data using BPSK.

    Figure 4.8 shows the flow chart for receiving data using BPSK modulation. The values of frequency and data rate should be specified at the command line while running the program. The frequency is specified based on the daughter board used. The frequency range of RFX400 daughter board is 400-500MHz. So we can specify a frequency in between 400MHz and
500MHz and it should match the frequency at the transmitter. Minimum data rate with which we can transmit the data using USRP is 32kbps. So we need to specify a data rate \( \geq 32 \text{kbps} \) and it must match the data rate at the transmitter. We need to setup the USRP board based on these specifications. In order to setup the USRP to transmit based on these specifications, we need to import program `usrp_receive_path` available in the gnuradio examples provided by the GNU radio package.

After setting up the USRP, if the receiver receives the packets, then we perform BPSK demodulation on this packet and remove the header and write the data in the packet to a file. Repeat this process till it receives the packets.

![Flow chart for transmission of data using BPSK.](image)

Figure 4.8. Flow chart for transmission of data using BPSK.
Further information on classes used to implement these blocks is provided in [18]. The information about how to write the signal processing blocks is given in [13]. The information regarding python language is provided in [14] [15]. If you have any issues regarding USRP and GNU radio refer [12].

4.4. Results

I used two USRP boards, one for transmitting and the other for receiving. I generated a file which contains the repeated sequence ‘10111’ and transmitted that file using BPSK modulation at frequency of 450MHz, data rate of 500kbps and a packet size of 1024. At the receiver, I received the packets and the packets are demodulated using BPSK demodulation. The extracted information is then stored in a file. I compared the contents of both the files and calculated the error rate. Figure 4.9 shows a picture of transmitter sending the packets using BPSK modulation. Figure 4.10 shows a picture of receiver receiving the packets. Table 4.1 shows the bit error rates (BER) for different values of transmission amplitude and data rates. The distance between transmitter and receiver is approximately 25m and they are out of line-of-sight. From the table, we can see that when the transmitter amplitude is reduced, the transmission power decreases and introduces errors into the file being transferred. One way to reduce the errors is to decrease the data rate. From the table, we can see that when there are errors, if we reduce the data rate, BER decreases. The other way to decrease the BER without changing the data rate is by introducing channel coding into the system and this is explained in chapter 5.
Figure 4.9. Picture of transmitter sending packets.

Figure 4.10. Picture of receiver receiving packets.
<table>
<thead>
<tr>
<th>Tx Amplitude(v)</th>
<th>Data Rate(kbps)</th>
<th>BER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01v</td>
<td>500</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>9mv</td>
<td>500</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>7mv</td>
<td>500</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>5mv</td>
<td>500</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>3mv</td>
<td>500</td>
<td>4.8 * 10^{-1}</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>1.28 * 10^{-1}</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>2.43 * 10^{-3}</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>1.2 * 10^{-4}</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.1. Bit Error rate for different values of transmission amplitude and data rates
CHAPTER 5
TURBO CODES IMPLEMENTATION

5.1. Introduction

Now-a-days our aim is to design a communication system which operates at low powers. If we consider the table 4.1, for a given data rate if the transmission amplitude (proportional to power) is reduced, the error rate increases after certain transmission amplitude. So the system we described in sections 4.2.3 and 4.3.3 is not suitable for low power operations. In order to convert this system to operate at low powers, we introduce channel coding which reduces the error rate. In this thesis, we are using the turbo codes to perform the channel coding. Turbo coding is an iterative soft-decoding algorithm that combines two or more convolutional codes and an interleaver to produce a block code that can approach Shannon limit. Figure 2.3 illustrates that the performance of BPSK scheme can be improved, based on the number of iterations in the decoder, by using turbo coding. So, by introducing the turbo codes we can improve the performance of the system discussed in sections 4.2.3 and 4.3.3 even when that system is operating at low powers.

5.2. Implementation

5.2.1. Transmitter

Figure 5.1 depicts the block diagram of the transmitter. First set the transmit signal path on the USRP by specifying the parameters such as frequency, data rate, packet size and the amplitude at which the signal must be transmitted by taking the input from the command line. The frequency is specified based on the daughter board used. The frequency range of RFX400 daughter board is 400-500MHz. So we can specify a frequency in between
400MHz and 500MHz. Minimum data rate with which we can transmit the data using USRP is 32kbps. So we need to specify a data rate \( \geq 32 \text{kbps} \). The data rate is limited by the execution time of the turbo decoder block (data rate = packetsize/execution time of the turbo decoder). To increase the data rate send the packets with some delay between them. Maximum packet size with which we can transmit using USRP is 4096. So we need to specify packet size \( \leq 4096 \). We need to setup a transmit signal path on the USRP board based on these specifications. In order to set up the USRP to transmit based on these specifications, we need to import program `usrp_transmit_path` available in the gnuradio examples provided by the GNU radio package. After that, read the data from the file that is to be transmitted and send it to the turbo encoder. The generator polynomial of the turbo encoder is given below

\[
\begin{align*}
1 & \quad 1 & \quad 1 \\
1 & \quad 0 & \quad 1
\end{align*}
\]

The rate of the turbo encoder which we are using is \( \frac{1}{2} \), i.e for every one bit input we will have two output bits for this turbo encoder. The output of the turbo encoder is twice the size of the data read from the file. The packet size must be specified based on the size of data after turbo encoding. We designed a turbo encoder which takes data of length 1024 at the input and gives an output data of length 2052. The last four bits in this data are not related to the information bits, they are used to terminate the trellis. The output of the turbo encoder is then modulated using BPSK modulation and transmitted using the USRP board and antenna.

![Figure 5.1. Block diagram of transmitter using turbo coding.](image)
5.2.2. Receiver

Figure 5.2 depicts the block diagram of the receiver. First set the receive signal path on the USRP by specifying the parameters such as frequency and data rate by taking the input from the command line. The frequency is specified based on the daughter board used. The frequency range of RFX400 daughter board is 400-500MHz. So we can specify a frequency in between 400MHz and 500MHz and it must match the transmitter frequency. Minimum data rate with which we can transmit the data using USRP is 32kbps. So we need to specify a data rate \( \geq 32\text{kbps} \) and it must match the transmitter data rate. We need to setup a receive signal path on the USRP board based on these specifications. In order to setup the USRP to receive based on these specifications, we need to import program `usrp_receive_path` available in the gnuradio examples provided by the GNU radio package. After that if the receiver receives the packets, then collect the received packets from USRP board and send it to demodulate block, to demodulate using the BPSK demodulation. The demodulated data is then sent to the turbo decoder. We set the number of iterations in the decoder to be 5. The turbo decoder decodes the received data and writes the extracted information data to a file and this process repeats till all the packets are received.

![Diagram](image)

Figure 5.2. Block diagram of receiver using turbo coding.

5.3. Results

I used two USRP boards, one for transmitting and the other for receiving. The file which contains the repeated sequence ‘10111’ is divided in to packets and transmitted using turbo coding and BPSK modulation, at frequency of 450MHz, data rate of 500kbps and a packet size of 2052. At the receiver, I received the packets and the packets are demodulated using
BPSK demodulation and then sent to turbo decoder (here the program for turbo decoding is written in C++ and interfaced to python using SWIG [16]). The extracted information is then stored in a file. I compared the contents of both the files and calculated the error rate. Figure 5.3 shows a picture of transmitter sending the packets using BPSK modulation and turbo coding. Figure 5.4 shows a picture of receiver receiving the packets and performs BPSK demodulation and turbo decoding. Table 5.1 shows the bit error rates (BER) for different values of transmission amplitude and data rates. The distance between transmitter and receiver is approximately 25m and they are out of line-of-sight. From the table, we can see that the system with turbo coding performs better than the system without turbo coding. The system with turbo coding performs similar to the system without turbo coding, in the absence of errors in system without turbo coding. But in the presence of errors, the system with turbo coding performs better than the system without turbo coding.

![Image of transmitter and receiver](image.png)

**Figure 5.3.** Picture of transmitter sending packets with turbo coding.
Figure 5.4. Picture of receiver receiving packets with turbo coding.
<table>
<thead>
<tr>
<th>Tx Amplitude(v)</th>
<th>Data Rate(kbps)</th>
<th>BER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01v</td>
<td>500</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>9mv</td>
<td>500</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>7mv</td>
<td>500</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>5mv</td>
<td>500</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>3mv</td>
<td>500</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.1. Bit Error rate for different values of transmission amplitude and data rates
In this demo, we will show that the performance is increased when turbo coding is used by transferring an audio file. To demonstrate this, we will select the system with BPSK modulation that operates in error prone region (by selecting a low transmission amplitude) and send an audio file through it and compare it with the system that includes turbo coding. For this demo, we use an audio file with sampling rate 48KHz and we use continuously variable slope delta modulation (CVSD) encoder to encode the audio samples. CVSD is used to encode speech that seeks to reduce the bandwidth required for transmission. It takes advantage of strong correlation between samples, quantizing the difference in amplitude between two consecutive samples. This requires fewer quantization levels as compared to other methods. It employs an adaptive algorithm that allows for continuous step size adjustment and a two level quantizer (one bit). In CVSD encoder, each incoming audio sample is compared to an internal reference value and if the input is greater or equal to the reference, the encoder outputs a ‘1’ bit, otherwise the encoder outputs a ‘0’ bit. The reference value is then updated accordingly based on the frequency of outputted ‘1’ or ‘0’ bits. The frequency with which we transmit is 450MHz and the packet size is 2052. The audio is sampled at 48KHz and each sample contains 8bits, so we need to have a bit rate of at least 384kbps. The data rate which we use is 400kbps and set the transmitter amplitude to 5mv.

6.1. System without Turbo Coding

Figure 6.1 depicts the block diagram of the transmitter which transmits an audio file. The audio file with .wav format is selected and used as a source (wave source). Then the audio file is encoded by making use of CVSD encoder. The output is then fed to packed-to-unpacked...
converter block which converts a stream of packed bytes to a stream of unpacked bytes. The output is then fed to the message sink which gathers the received items into messages and insert into message queue. The data is collected from the message queue and converted to packets. The data in the packets is then modulated using BPSK modulation and transmitted using the USRP board.

![Figure 6.1. Block diagram of transmitter to transmit audio file.](image)

Figure 6.1. Block diagram of transmitter to transmit audio file.

Figure 6.2 depicts the block diagram of the receiver which receives an audio file. The received packet is collected from the USRP board and demodulated using BPSK demodulation. The demodulated data is then converted into stream of data and sent to unpacked-to-packed converter block which converts a stream of unpacked bytes to stream of packed bytes. The output is then decoded using CVSD decoder and sent to the audio sink which plays the audio file. The received audio file contains noise.

![Figure 6.2. Block diagram of receiver to receive audio file.](image)

Figure 6.2. Block diagram of receiver to receive audio file.

6.2. System with Turbo Coding

Figure 6.3 depicts the block diagram of the transmitter which transmits an audio file with turbo coding. The audio file with .wav format is selected and used as a source (wave source). Then the audio file is encoded by making use of CVSD encoder. The output is then fed to packed-to-unpacked converter block which converts a stream of packed bytes to a stream of
unpacked bytes. The output is then fed to message sink which gathers the received items into messages and insert into message queue. The data is collected from the message queue and converted to packets. The data in the packets is encoded using turbo encoder and the encoded data is then modulated using BPSK modulation and transmitted using the USRP board.

![Figure 6.3. Block diagram of transmitter to transmit audio file using turbo coding.](image)

Figure 6.3. Block diagram of transmitter to transmit audio file using turbo coding.

Figure 6.4 depicts the block diagram of the receiver which receives an audio file with turbo coding. The received packet is collected from the USRP board and demodulated using BPSK demodulation. The demodulated data is then decoded using turbo decoder and then converted into stream of data and sent to unpacked-to-packed converter block which converts a stream of unpacked bytes to stream of packed bytes. The output is then decoded using CVSD decoder. Since the turbo decoder introduces a delay, instead of directly sending the output of CVSD decoder to audio sink, we create a buffer to store the audio samples and write the audio samples to audio sink when the buffer is full. We write audio samples into buffer while reading the buffer from the other side. The received audio file plays with certain initial delay due to buffer but it eliminates noise.

![Figure 6.4. Block diagram of receiver to receive audio file using turbo coding.](image)

Figure 6.4. Block diagram of receiver to receive audio file using turbo coding.
CHAPTER 7

CONCLUSION AND FUTURE WORK

In this thesis, GNU radio is explored with the objective to implement turbo coding and show that the performance can be improved by making use of turbo codes. From the experiments, we can conclude that, GNU radio along with USRP can provide immense implementation flexibility but the hardware components used currently have constraints in terms of packet size, data rates and processing speed. We also observed that the real-time implementation of transferring audio file using turbo coding is constrained by the decoder latency. As the execution time of the decoder increases, the initial delay before playing the audio file at the receiver increases.

As future work, the receiver can be improved to reduce the initial delay at the receiver while playing the audio file and also implementing the system at even higher frequencies. We can also design a system which transfers an image or a video by using turbo codes.
BIBLIOGRAPHY


[16] Interfacing c++ and python using SWIG. Available at http://www.penzilla.net/tutorials/python/swig/.


