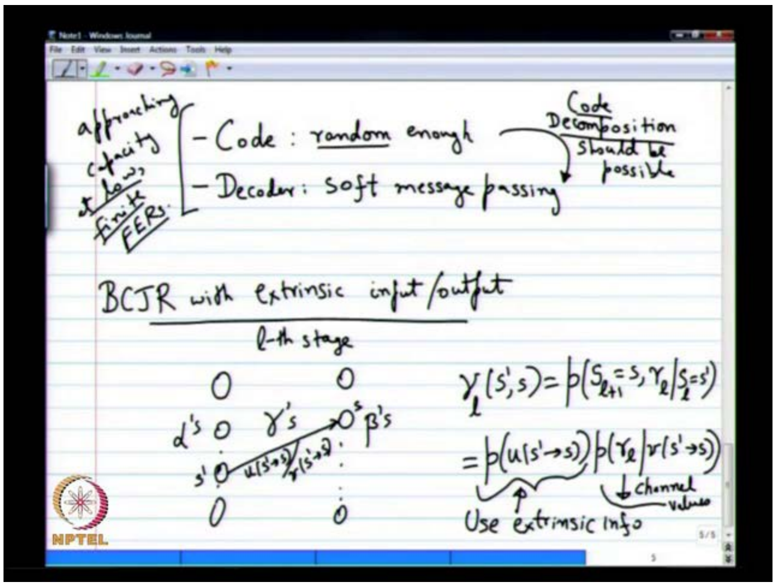
Coding Theory

Prof. Dr. Andrew Thangaraj
Department of Electronics and Communication Engineering
Indian Institute of Technology, Madras

Lecture - 37
Turbo Codes in Practice

(Refer Slide Time: 00:14)



approaching capacity at low, ~~finite FERs~~

- Code : random enough → Code Decomposition should be possible
- Decoder: soft message passing

BCJR with extrinsic input/output

l-th stage

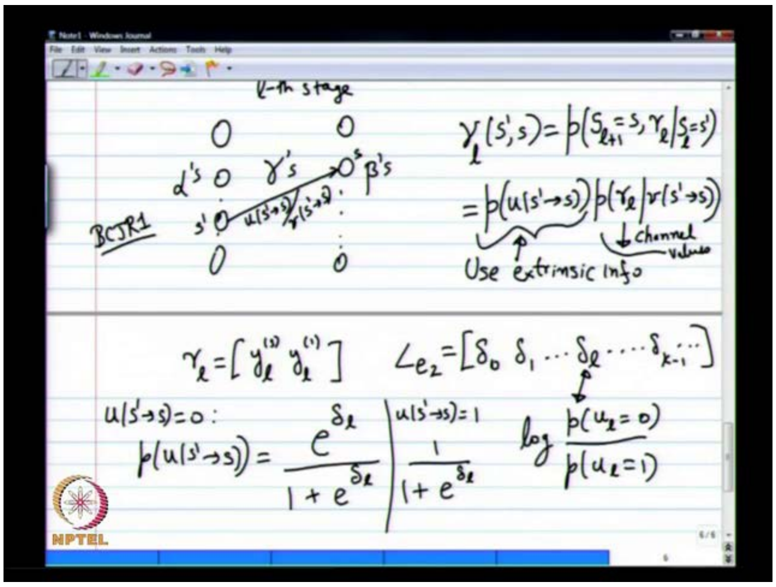
$$\gamma_l(s',s) = p(S_{l+1}=s, r_l \mid S_l = s')$$

$$= \underbrace{p(u(s'\to s))}_{\text{Use extrinsic info}} \, \underbrace{p(r_l \mid v(s'\to s))}_{\text{channel values}}$$

So, here is where we were, we were looking at the BCJR algorithm with extrinsic input and output and how does it work. And looking at the expression it is clear that the second part, so once again I think there was a question about what this u and v are? So, basically if you have a, if you have a branch, there will be a label on the branch. This is s prime and this is s and the label on the branch is basically s prime s that is the input and stroke and there will be an output.

The output is basically v of s prime s. So, that is my notation when I say u and v. And clearly the second part here does not use any extrinsic information, it is just it uses only channel, uses channel values. The first part is where we have to really use the prior information. So, how do we do this? It is not too hard to imagine how you can do this. If you get if you look at the l t h stage, so I think the picture is somehow every time I do this. The picture gets to the bottom of the screen and let me see like I can do this also. So, what is happening in our turbo kind of situation?

(Refer Slide Time: 01:37)



$$\gamma_l(s',s) = p(S_{l+1}=s, r_l \mid S_l = s')$$

$$= \underbrace{p(u(s'\to s))}_{\text{Use extrinsic info}}\, \underbrace{p(r_l \mid v(s'\to s))}_{\text{channel values}}$$

$$r_l = [y_l^{(s)}\ y_l^{(1)}] \qquad L_{e_2} = [\delta_0\ \delta_1 \cdots \delta_l \cdots \delta_{k-1} \cdots]$$

$u(s'\to s)=0$:

$$p(u(s'\to s)) = \frac{e^{\delta_l}}{1+e^{\delta_l}} \quad \Bigg| \quad u(s'\to s)=1 \quad \frac{1}{1+e^{\delta_l}} \qquad \log\frac{p(u_l=0)}{p(u_l=1)}$$

So, you have so this is maybe BCJR 1. So, if this is BCJR 1 your r l is going to be what, y s l and y 1 l that is, this is BCJR 1. If this is BCJR 2 then this is going to be y s l will be the same and then it will be y 2 l, that is r l and v and u are kind of going to be the, so u of s prime s. So, those things depend only on the branch. So, you cannot say much about it and what is the extrinsic information coming in?

So, you would have had l e 1, l e 1 or l e 2, so l e 2 which will be let us say just for simplicity of notation. I will simply say l 0 l 1 all the way to I do not want to use l because I have to use subscript l also. What we can use here b delta, delta 1 delta 0 delta 1. So, on you will also have a delta l all the way to delta let us say k minus 1 maybe. So, that is how l e 2 will look and this is basically the extrinsic information. And what is this, how should we think of this?

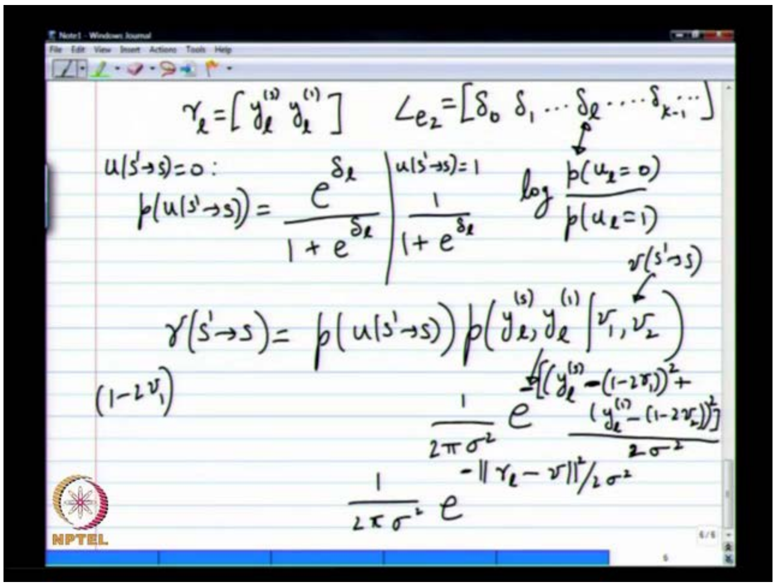
This is basically log of a ratio a ratio of what? Probability of U l equals 0 divided by probability that u l equal to 1. So, all that is given that, but this basically I am going to think of the extrinsic LLR as some log likelihood ratio and remember this, this is extrinsic. So, this probability basically means, it does not use anything else, so it is extrinsic. Now if u of s prime s is 0 then, what I should put here, I have to compute p of u l equal to 0 and then substitute that there.

So, form here easily you can see that a priori probability for u of s prime s equal to 0 for a branch, which corresponds to u of s prime s equal to 0. What will be p of u s prime s; it will be e power delta l. What will it be?

Student: ((Refer Time: 04:12))

You can show this quite easily. So, p of u l equal to 0 is e power delta l time's p of u l equal to 1 and if you add those 2 up it should be equal to 1. So, you should put that in you are getting at this is that e power delta l. On the other hand if u of s prime is equal to 1 then, what will be this 1 by 1 plus e power delta l. So, let me write that down also here. If this is equal to 1 this k becomes 1 by 1 plus e power delta l. So, that is how you use the extrinsic information. So, it is easy enough to use the extrinsic information.

(Refer Slide Time: 05:06)



$$r_l = [y_l^{(s)} \; y_l^{(1)}] \qquad L_{e_2} = [\delta_0 \; \delta_1 \cdots \delta_l \cdots \delta_{k-1} \cdots]$$

$u(s' \to s) = 0:$

$$p(u(s' \to s)) = \frac{e^{\delta_l}}{1 + e^{\delta_l}} \quad \Bigg| \quad u(s' \to s) = 1 \quad \frac{1}{1 + e^{\delta_l}} \qquad \log \frac{p(u_l = 0)}{p(u_l = 1)}$$

$v(s' \to s)$

$$\gamma(s' \to s) = p(u(s' \to s))\, p(y_l^{(s)}, y_l^{(1)} \mid v_1, v_2)$$

$(1 - 2v_1)$

$$\frac{1}{2\pi\sigma^2} e^{-\frac{[(y_l^{(s)} - (1-2v_1))^2 + (y_l^{(1)} - (1-2v_2))^2]}{2\sigma^2}}$$

$$\frac{1}{2\pi\sigma^2} e^{-\|r_l - v\|^2 / 2\sigma^2}$$

So, how do you compute the extrinsic output is the next question? So, for that we have to look at gamma little bit more closely. So, basically we are saying gamma of s prime s is nothing but p of u of s prime s. Here is where we use the extrinsic information time's p of r l, which is p of y s l comma y 1 l given, given what? Given v of s prime s, so I have to split it into 2, so how do I do that? I can do it this way. So, I mean it does not matter. So, s is showing up in too many places, so I will simply say v 1 v 2. So, it is for s prime s prime to s, so I am just swallowing the notation here, this is basically v of s prime s.

So, what will this be? This will basically be 1 by 2 pi sigma square e power y s l minus, minus what? So, to write v 1 I have to basically met make v 1 and v 2 the v p s k encoded versions, so what can we do there, 1 minus 2 v 1. So, do I have a notation for that I think?

Student: ((Refer Time: 06:41))

Y is basically my received value, so v 1 v 2 is basically now in bits 0 or 1. And when I write minus here I have to write 1 minus 2 v 1, so maybe we can let 1 minus 2 v 1 to be what w 1.

Student: ((Refer Time: 07:05))

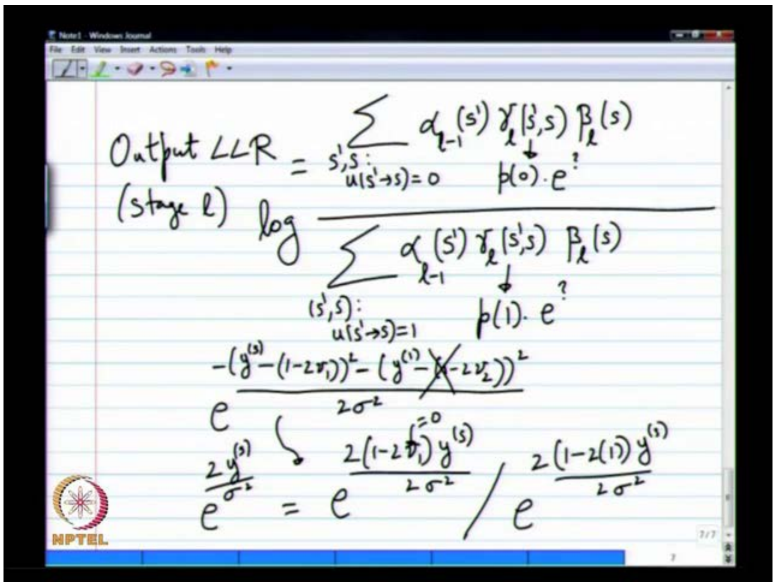
Somebody could have told me. So, we had r l and r l 1, but anyway it does not matter. So, it is hopefully it is clear to you. Maybe for the recording I might have to do redo this lecture with a different notation, but anyway, so 1 minus 2 v 1 is basically the BPSK encoded version of v 1. So, there is also another way to write this, I think you can also write it as minus 1 power v 1. So, anyway all these things do not matter. So, I am just trying to think of a very easy notation for this. So, what can we do any ideas?

Student: ((Refer Time: 07:51))

As minus 1 plus 1 that is too many. So, me let me just keep it as 1 minus 2 v 1, so it is not too bad square plus what y l 1 minus 1 minus 2 v 2 square. Of course, this everything is divided by there is a minus overall also, this is the formula. So, maybe there is a there is a neater way of writing it, so usually what people do is, they would write like this. So, 1 by 2 pi sigma square e power minus they will put a 2 bars and that means like norm.

And then what they would do is they simply write r l minus v square by 2 sigma square. So, that is also some other way of writing it, but anyway. So, it is just it is irrelevant, so that is the form for the other part. So, clearly the channel received values enter into the second term and the a priori values extrinsic values enter into the first term. So, that is the main lesson here.

(Refer Slide Time: 09:09)



Output LLR (stage l) $= \log \dfrac{\displaystyle\sum_{\substack{s',s:\\ u(s'\to s)=0}} \alpha_{l-1}(s')\, \underset{\overset{\downarrow}{p(0)\cdot e^{?}}}{\gamma_l(s',s)}\, \beta_l(s)}{\displaystyle\sum_{\substack{(s',s):\\ u(s'\to s)=1}} \alpha_{l-1}(s')\, \underset{\overset{\downarrow}{p(1)\cdot e^{?}}}{\gamma_l(s',s)}\, \beta_l(s)}$

$$e^{\frac{-(y^{(s)}-(1-2v_1))^2-(y^{(1)}-\cancel{(1}-2v_2))^2}{2\sigma^2}}$$

$$e^{\frac{2y^{(s)}}{\sigma^2}} = e^{\frac{2(1-2\overset{=0}{v_1})y^{(s)}}{2\sigma^2}} \Big/ e^{\frac{2(1-2(1))y^{(s)}}{2\sigma^2}}$$

So, once you have that we can now go ahead and compute the final LLR. So, what is the output LLR for stage 1? For stage l in stage l, you are going to now add up over s prime s, such that u of s prime s equals 0 then, you divide with summation s prime s such that u of s prime s equals 1. So, there is log also, I am sorry log and then what happens there is you have an alpha term and then you have a gamma and then you have a beta term.

So, let us write that down, there is going to be an alpha l minus 1 of s prime gamma l s prime s and then beta l s. And then you have an alpha l minus 1 s prime gamma l s prime s, beta l s. So, whether it is intrinsic or extrinsic or anything at all, all that is going to be part of only gamma. Alpha and beta will generally multiply anything that you put in there. So, we have to only substitute for gamma. What are we going to put for gamma? Here I am going to put p of u s prime s, what is that in the numerator u s prime s is always 0.

So, it is going to be simply p of the a priori 0 times some e power term, so e power something. So, let us keep it like that. In that e power something there will be 2 things, so I will come to that later. So, here you have what p of 1 times e power something. So, already you see there is something that is emerging some. There is some simplicity here, so p of 0 there is one simplification that is that is already kind of possible here. So, we will we will come to that. So, there are some common things that will come out, so p of 0 by p of 1 will come out.

So, log p of 0 by p of 1 is anyways is the extrinsic, so the extrinsic is coming out by an as an independent factor. Now, we have to see if there is anything else that will come out as an independent factor. What do you what else do you expect as an independent as a factor that is going to come out? So, the extrinsic has come out already p of 0 by p of 1 and from the e power part what do you want to come out?

Anything that depends on the y s part has to come out then, I want to find the extrinsic output what am I trying to find? I am trying to get rid of the l e 2 part that I am able to already get rid of; it is just the first simplification. Next I have to get rid of whatever depends on y s, the question is will anything that depends on y s come out of that gamma, the branch matrix. So, p of 0 by p of 1 anyway comes out and then I have to look that what is in the exponential.

So, let us look at what is in the exponential? The exponential will have minus; I will simply it as y s so, it is maybe clear y s minus 1 minus 2 v 1 square minus y 1 minus 1 minus 2 v 2 squares by 2 sigma squares. So, this is the expression that we had. Now, if we try and simplify what is going to happen, what are the various things that are going to happen? y a square is going to come, but there will be a similar y a square term in the denominator and that is going to cancel with this. And then what else will you have you will have a 1 minus 2 v 1 square what will be that that will always be 1.

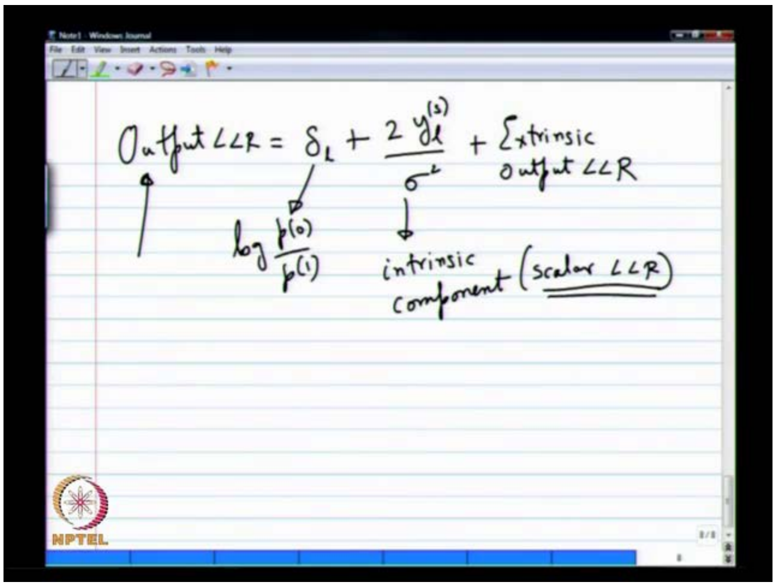
Similarly, in the denominator whatever v 1 and v 2 is that is always going to be 1 and that is also going to cancel. So, the only thing that will be retained that you have to retain here is 2 y s times 1 minus 2 v 1 by 2 sigma square. So, you will have a similar term in the denominator also and that you have to read. Similarly, in the second part, you will have a y 1 square, in the denominator also you will have a y 1 square term and then 1 minus 2 v 2 square will also cancel with what is in the denominator.

What you have to retain is 2 times 1 minus 2 v 2 y 1 by sigma square. Remember this v 1 and v 2 are not the same for each term, they change depend on the depending on what the term is. v 1 and v 2 are not the same. So, what happens in the recursive systematic encoder? V 1 becomes equal to u, so the u s prime s is 0 means v 1 is always 0 in the numerator and v 1 will be always 1 in the denominator that is the thing only I know. v 2 on the other hand will be arbitrary and it will change and all that, but anyway v 2 I do not care too much about because that is only getting involved with the parity part.

I want to only bring out the message systematic received value. So, ignoring this part not worrying too much about this, if you look at this term it turns out the only thing that this simplifies to e power 2 1 minus 2 v 1 times y s by 2 sigma square. And v 1 is going to be 0 for the numerator and it is going to be 1 in the denominator in my recursive systematic encoder. So, what will happen finally when I take the ratio, this divided by e power so I have to put v 1 equal to 0 here then e power 2 1 minus 2 times 1 y s by 2 sigma square.

So, finally, all these things add up to what e power 2 y s by sigma square. So, what comes out? There are two things that come out common in this ratio. One is p of 0 by p of 1 the other is e power 2 y s by sigma square. So, if you do this simplifications, I know I have not done it in very good detail, but go through and look at it and think about all the think about each of this steps it is not too difficult. So, the first part is p of 0 and p of 1 the second part what subtracts from y of s is always either plus 1 in the it is always plus 1 for the numerator and minus 1 for the denominator. So, you can pull out the y s part separately.

(Refer Slide Time: 16:58)



$$\underset{\uparrow}{\text{Output LLR}} = \underset{\swarrow\ \log\frac{p(0)}{p(1)}}{\delta_l} + \underset{\downarrow\ \text{intrinsic component (scalar LLR)}}{\frac{2y_l^{(s)}}{\sigma^2}} + \text{Extrinsic output LLR}$$

When you do that, you see that the final output LLR simplifies to delta l. How did I get this? This is basically log p of 0 by p of 1 that is this is delta l plus two times y s l by sigma square plus something else and what that something else will be, the extrinsic output. So, how I am running the BCJR algorithm, now with the extrinsic input the remember there were like 4 or 5 steps in the BCJR algorithm.

First step was computing the branch matrix, second step was doing the forward recursion computing all the alphas, third step was doing the backward recursion computing, computing all the betas and then, final step was computing the LLRs alpha beta gamma, alpha gamma beta, alpha gamma beta type of thing. Now, with extrinsic input the first change is when you compute your branch matrix, you have to incorporate the extrinsic input in this form.

So, your branch matrix computation will change. Is there is anything change in the second step, absolutely nothing. What about the third step, computing betas absolutely nothing, everything is the same. What about the fourth step, computing the total output LLR exactly the same there is no problem, after you compute the total output LLR, which is what comes up here on the left hand side. At the end of the fourth step, you would have computed the total output LLR.

From there how will you compute the extrinsic output LLR, you subtract the intrinsic extrinsic input delta l and then, you subtract this other factor 2 y l s by sigma square, so this is the intrinsic parts? When you subtract these two, you are going to get the extrinsic output LLR which needs to be sent to the second decoder. This is the scalar LLR for the intrinsic component. So, this is like the scalar LLR that we had before.

Student: Completely eliminating the extrinsic input because the alpha is also to some extent alphas and betas also depend on.

Well it is not really true. See look at, if you are saying.

Student: Eliminating from the previous branch matrix which actually.

So, you are saying the extrinsic? You have to eliminate the extrinsic input in the l th stage, the extrinsic input from the other stages can filter through into your l th stage. So, the question here is interesting and you have to pay attention to the question. The question is we are saying we are subtracting the extrinsic, but that is on a stage by stage basis.

You do not totally subtract the extrinsic, you should not be totally subtracting it out because there is a trellis dependency on this information and you have to exploit this that trellis dependency. So, in your l th stage the other extrinsic will feed into it, the other channel information will also feed into it. All those you are not going to subtract from the l th stage. In the l th stage you are going to subtract the extrinsic you received for that stage and the channel systematic received value for that particular state only. So of course, not just the extrinsic even the y s that you receive for l minus 1 will feed into this.

All that you do not subtract that you will use, but for in that l th stage explicitly you subtract what is coming through. So, that is I mean if you think about it that is an approximation in your iterations, so there is some dependency that will carry through. You are not really subtracting all the dependency, but if you account for all of that then your decoder is becoming optimal. You do not want to do optimal because it is going to become really huge; this is not going to be implementable.

So, that is essentially the approximation you are doing. So, you feed in something, but since it is through some other path eventually, the dependence will hurt you only after a few iterations. So, if you have converged already by a few iterations you are happy that is the idea, but that is an important to keep in mind. You do not throw away all the extrinsic, but intrinsic only that in corresponding to the stage. Now, we have filled out all the necessary details for the turbo decoder, which is here. Once again I am sorry for this notation I guess, I wish somebody had pointed out that this should be r instead of y.

So, y was after the BPSK, so I think here I made the mistake, mistake came here. So, after BPSK it should have been.

Student: ((Refer Time: 21:59))

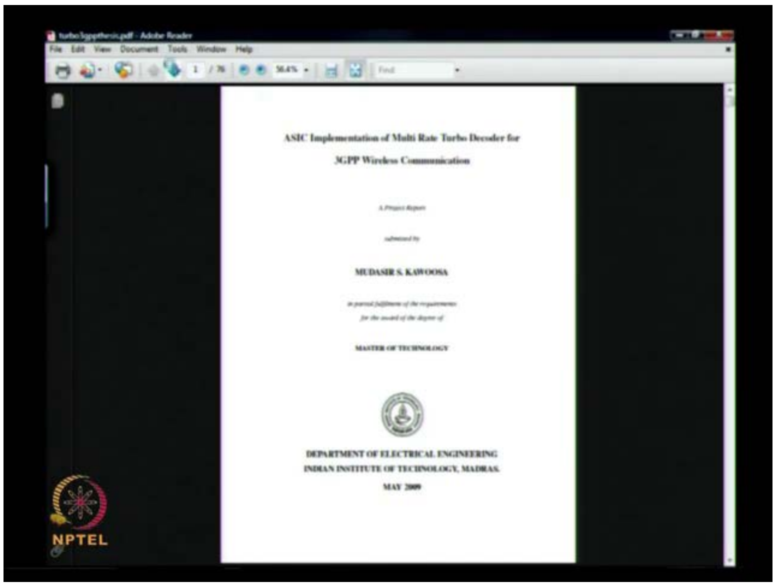
BPSK plus a w g n is how I am using. I should not have done that. I should have said after BPSK it is y, after a w g n it is r. If I had done that and everything would have been very hunky again, but anyway it went off for a toss that is how it is hopefully you too much. Now, we have all these inputs, so if you had taken the pains to implement the BCJR algorithm like I have asked you to do last time.

Maybe not everyone if you did it then, you can now modify that implementation to add using extrinsic and computing extrinsic. Once you do that you can even implement the turbo decoder. So, it is not too scary to implement, if you write if you are good in Mat lab and you write a Mat lab code for this, it will hardly take about 40-50 lines. Finish off your turbo decoder implementation that is all it takes to achieve or please approach capacity with finite f e r.

Any questions on any of these computations, so one question to ask is this is very much a BPSK computation. If you look at it I have strongly used some BPSK properties. What happens if it is not BPSK etcetera, those are questions that you can ask, there are things you can do. So, to work around that, but today what you do is how to get down to this kind of a thing is to kind of approximate things like so that they look like BPSK and then you try and work around and so that is the idea.

So, usually I mean there are some additional tricks needed to deal or deal with arbitrary q m consolations, but we would not go into that BPSK is at least straight forward. So, what we are going to do next is, maybe see a few more. So, I had several students worked on implementation of turbo and LDPC codes on in hardware. And there is a couple of thesis particularly from last year's or year before that two M tech. students who wrote, who did a very good job with implementing LDPC and turbo codes in VLSI. So, what I am going to show, you see there is a turbo 3 g p p thesis that is from some student name Mudassir 2009.

(Refer Slide Time: 24:50)


ASIC Implementation of Multi Rate Turbo Decoder for

3GPP Wireless Communication

A Project Report

submitted by

MUDASIR S. KAWOOSA

in partial fulfillment of the requirements

for the award of the degree of

MASTER OF TECHNOLOGY

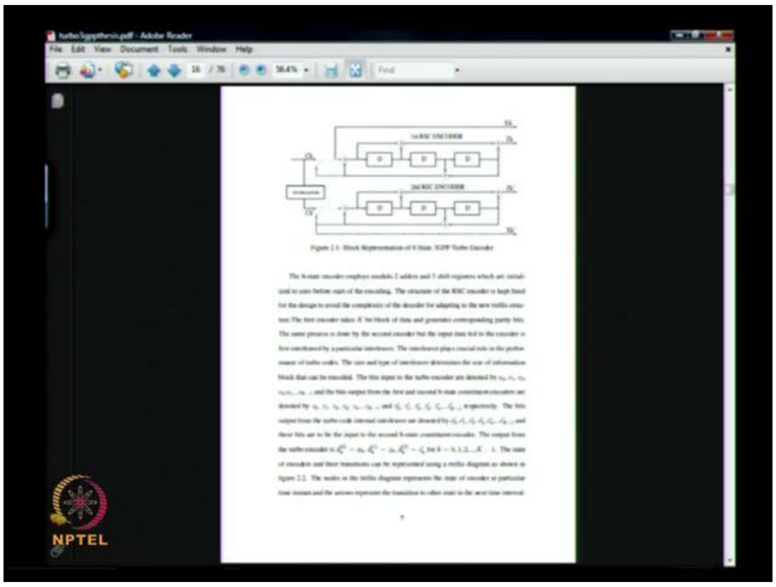
DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.

MAY 2009

So, we are not going to look at this entire report, so let us go through the main part of part I want to show you.

(Refer Slide Time: 25:13)



turbo3gppthesis.pdf - Adobe Reader
File Edit View Document Tools Window Help
1st RSC ENCODER
2nd RSC ENCODER
INTERLEAVER
D

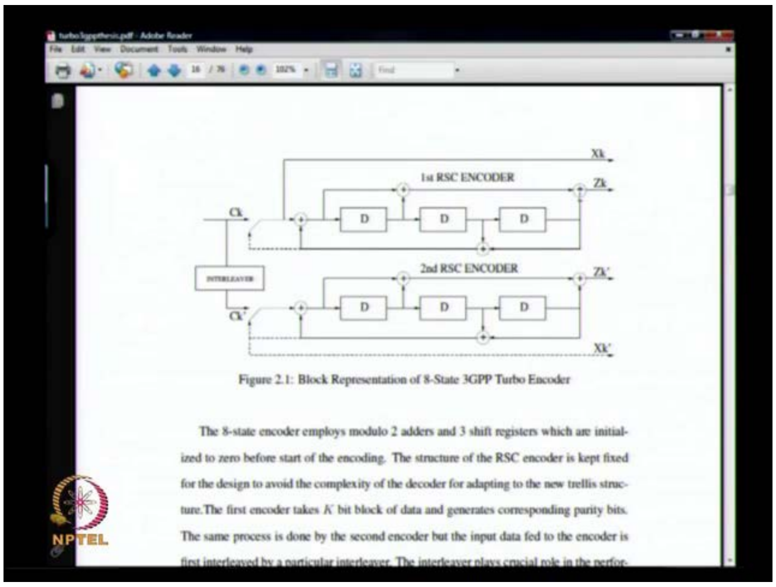
Figure 2.1: Block Representation of 8-State 3GPP Turbo Encoder

The 8-state encoder employs modulo 2 adders and 3 shift registers which are initialized to zero before start of the encoding. The structure of the RSC encoder is kept fixed for the design to avoid the complexity of the decoder for adapting to the new trellis structure. The first encoder takes K bit block of data and generates corresponding parity bits. The same process is done by the second encoder but the input data fed to the encoder is first interleaved by a particular interleaver. The interleaver plays crucial role in the performance of turbo codes. The size and type of interleaver determines the size of information block that can be encoded. [illegible]

7

So, this is the 8th state 3 g p p turbo encoder. So, 1 plus d plus d power 3 by 1 plus d square plus d power 3. You can see the feedback is 1 plus d square plus d power 3 and what is taken out is 1 plus d plus d power 3.

(Refer Slide Time: 25:43)



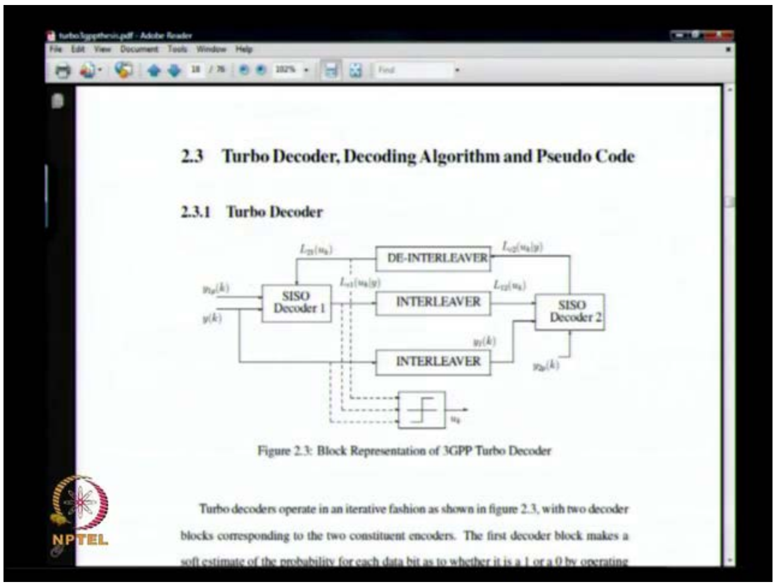
turbo3gppthesis.pdf - Adobe Reader
File Edit View Document Tools Window Help
Xk
1st RSC ENCODER
Zk
Ck
D
D
D
INTERLEAVER
2nd RSC ENCODER
Zk'
Ck'
D
D
D
Xk'

Figure 2.1: Block Representation of 8-State 3GPP Turbo Encoder

The 8-state encoder employs modulo 2 adders and 3 shift registers which are initialized to zero before start of the encoding. The structure of the RSC encoder is kept fixed for the design to avoid the complexity of the decoder for adapting to the new trellis structure.The first encoder takes K bit block of data and generates corresponding parity bits. The same process is done by the second encoder but the input data fed to the encoder is first interleaved by a particular interleaver. The interleaver plays crucial role in the perfor-

Then you have the Intel lever so, this is the picture. So, this what he implemented he put this so, these days I think I do not know how many of you do v log v h d l or anything like that. It is almost like c programming. Write a program and press compile it becomes an f p g a implementation. This is something like that, but of course, he did a lot more stuff than that, but that is one way of thinking about it. So, that is the trellis there are some things, so in fact they terminate both actually.

(Refer Slide Time: 26:15)



2.3 Turbo Decoder, Decoding Algorithm and Pseudo Code

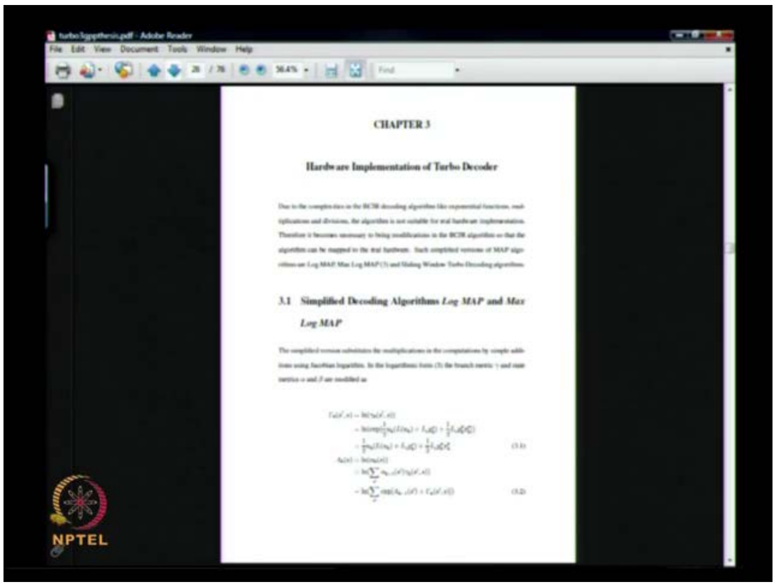
2.3.1 Turbo Decoder

Figure 2.3: Block Representation of 3GPP Turbo Decoder

Turbo decoders operate in an iterative fashion as shown in figure 2.3, with two decoder blocks corresponding to the two constituent encoders. The first decoder block makes a soft estimate of the probability for each data bit as to whether it is a 1 or a 0 by operating

So, there is a picture of how the encoder would look in all that extrinsic, intrinsic etcetera and then, the turbo decoding algorithm and that all kinds of pseudo code. So, let me quickly go to the part that I want to show you, the hardware implementation.

(Refer Slide Time: 26:37)



CHAPTER 3

Hardware Implementation of Turbo Decoder

Due to the complexities in the BCJR decoding algorithm like exponential functions, multiplications and divisions, the algorithm is not suitable for real hardware implementation. Therefore it becomes necessary to bring modifications in the BCJR algorithm so that the algorithm can be mapped to the real hardware. Such simplified versions of MAP algorithms are Log MAP, Max Log MAP [3] and Sliding Window Turbo Decoding algorithms.

3.1 Simplified Decoding Algorithms *Log MAP* and *Max Log MAP*

The simplified version substitutes the multiplications in the computations by simple additions using Jacobian logarithm. In the logarithmic form [3] the branch metric γ and state metrics α and β are modified as

[illegible] (3.1)

[illegible] (3.2)

So, there is lots of stuff you have to take care of when you implement. So, you see this max star versus max. So, you can see the formula that is there.

(Refer Slide Time: 26:56)

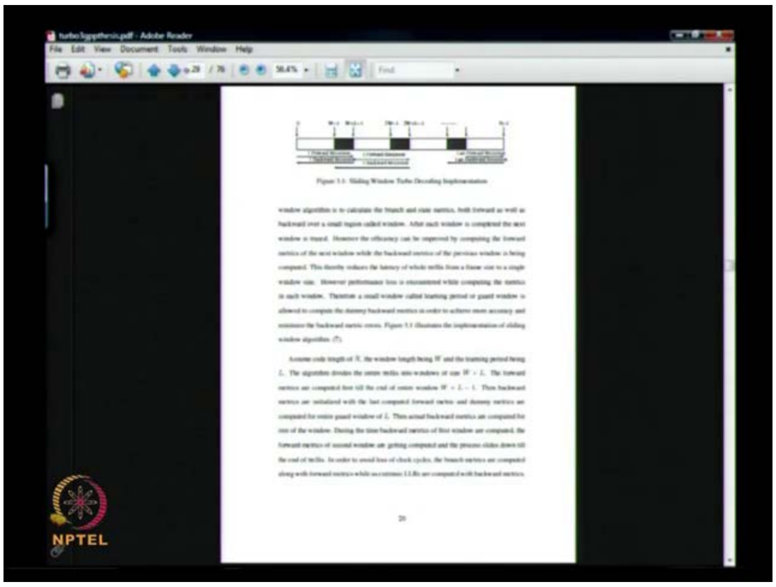


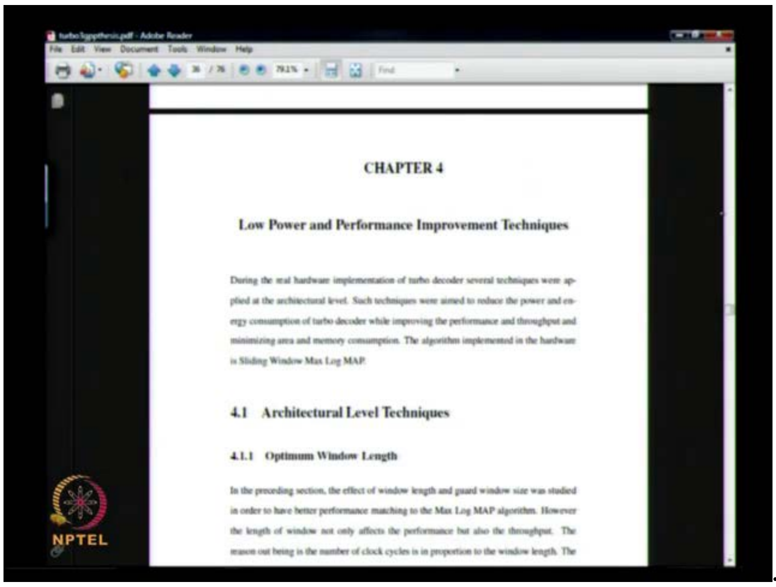
Figure 3.1: Sliding Window Turbo Decoding Implementation

[illegible]

20

So, hardware implementation let me show you, there is like a this is first the windowing part. So, he is first talking about the sliding window max log map. So, I will come to this maybe a little bit later. So, let us look at performance comparison this is quite important.

(Refer Slide Time: 27:01)


CHAPTER 4

Low Power and Performance Improvement Techniques

During the real hardware implementation of turbo decoder several techniques were applied at the architectural level. Such techniques were aimed to reduce the power and energy consumption of turbo decoder while improving the performance and throughput and minimizing area and memory consumption. The algorithm implemented in the hardware is Sliding Window Max Log MAP.

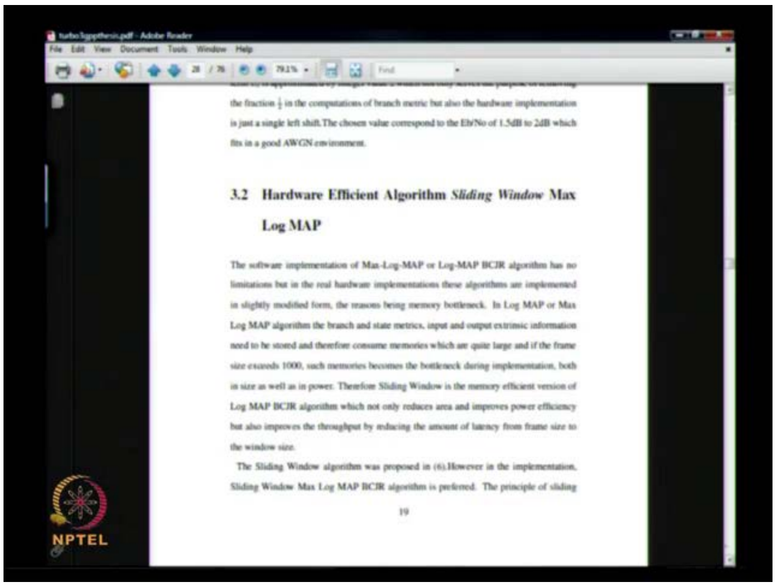
4.1 Architectural Level Techniques

4.1.1 Optimum Window Length

In the preceding section, the effect of window length and guard window size was studied in order to have better performance matching to the Max Log MAP algorithm. However the length of window not only affects the performance but also the throughput. The reason out being is the number of clock cycles is in proportion to the window length. The

It is really it took off; it is really upset with me for something. What too sensitive computers are becoming more and more human sensitive you know.

(Refer Slide Time: 27:59)



the fraction $\frac{1}{2}$ in the computations of branch metric but also the hardware implementation is just a single left shift.The chosen value correspond to the Eb/No of 1.5dB to 2dB which fits in a good AWGN environment.

3.2 Hardware Efficient Algorithm *Sliding Window* Max Log MAP

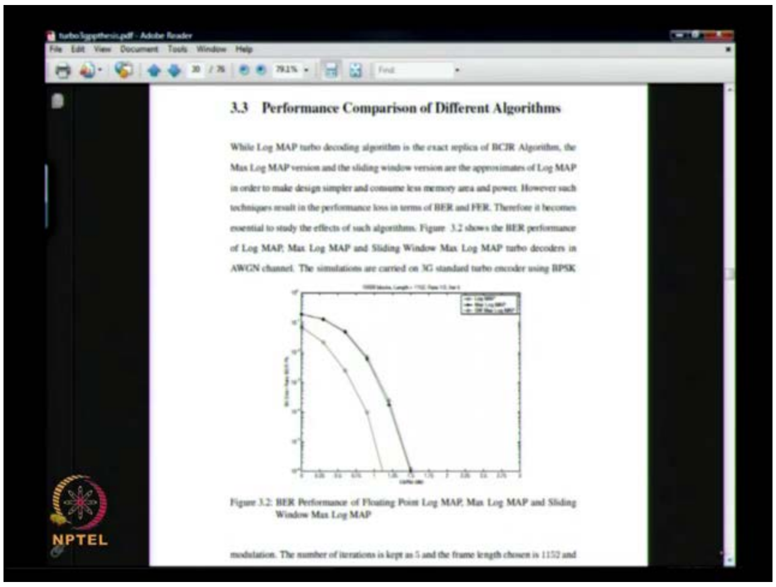
The software implementation of Max-Log-MAP or Log-MAP BCJR algorithm has no limitations but in the real hardware implementations these algorithms are implemented in slightly modified form, the reasons being memory bottleneck. In Log MAP or Max Log MAP algorithm the branch and state metrics, input and output extrinsic information need to be stored and therefore consume memories which are quite large and if the frame size exceeds 1000, such memories becomes the bottleneck during implementation, both in size as well as in power. Therefore Sliding Window is the memory efficient version of Log MAP BCJR algorithm which not only reduces area and improves power efficiency but also improves the throughput by reducing the amount of latency from frame size to the window size.

The Sliding Window algorithm was proposed in (6).However in the implementation, Sliding Window Max Log MAP BCJR algorithm is preferred. The principle of sliding

19

So, this is what I wanted to show you.

(Refer Slide Time: 28:05)



3.3 Performance Comparison of Different Algorithms

While Log MAP turbo decoding algorithm is the exact replica of BCJR Algorithm, the Max Log MAP version and the sliding window version are the approximates of Log MAP in order to make design simpler and consume less memory area and power. However such techniques result in the performance loss in terms of BER and FER. Therefore it becomes essential to study the effects of such algorithms. Figure 3.2 shows the BER performance of Log MAP, Max Log MAP and Sliding Window Max Log MAP turbo decoders in AWGN channel. The simulations are carried on 3G standard turbo encoder using BPSK

Figure 3.2: BER Performance of Floating Point Log MAP, Max Log MAP and Sliding Window Max Log MAP

modulation. The number of iterations is kept as 5 and the frame length chosen is 1152 and

So, the first thing to keep in mind is when I write down the algorithm in the way I write it down, I am treating everything as a real number. I am treating everything as a real number, but real numbers do not exist. There is no such thing as a real number only it exists inside your head and you can work with it, but in practice there is no real number. In computation you can either use pre finite precision or floating point, so those are the only things you can use.

Only a finite number of numbers you can deal. So, the first thing is to do a floating point implementation. Why is floating point the easiest to do? You can write a c program or a Mat lab program, it will naturally do floating point. If you wanted to do fixed point finite precision, you have to do some additional effort to make it fixed point finite precision. So, a floating point is first thing you implement. So, you see for instance this plot that he has here this is for the if I do this will it go up.

No it does not go up it just go up. So, let me zoom into this plot and show you. Can you see that is that good? So, that is the first plot I am showing you. So, this is bit error rate versus e b o r n naught for a rate 1 by 3 turbo code with 5 iterations. You can see that in the title and the length is 1152 that is the maximum possible length in 3 g p p and the assimilated 10 thousand blocks. And you can see here, there is there are three different decoders, so one is called log map, a log map is basically the there is the basically BCJR complete BCJR.

Next is max log map, there instead of max star you do max. So, you do not use the sigma square there is lots of benefit and that. So, you see there is about like a if you look at it loss is 10 power minus 6 bit error rate, what is the loss roughly about 0.4 sort of maybe 0.4 d b loss from log map to max log map. And then he has a s w, s w is basically a sliding window. So, 1152 is a long length you cannot store even if it is fixed point width 8 bit precision or 10 bit precision that is a lot of bits to store for alphas. So, you have to do sliding window and that is where he had this picture. I thought that was a good picture to show how the sliding window works. So, you can see the picture on the top.

(Refer Slide Time: 30:12)

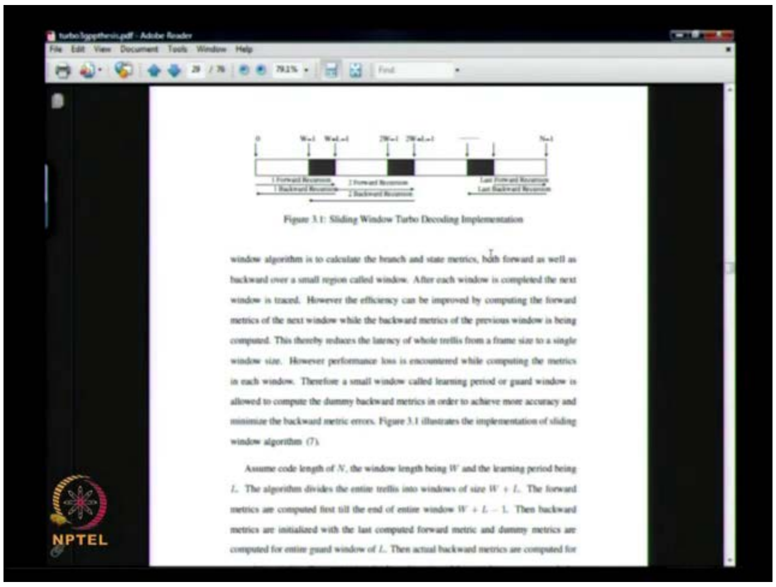


Figure 3.1: Sliding Window Turbo Decoding Implementation

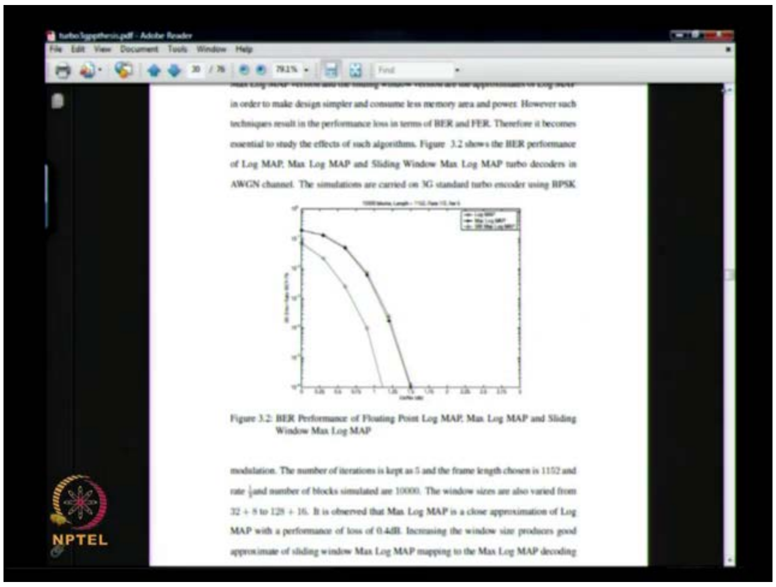
window algorithm is to calculate the branch and state metrics, both forward as well as backward over a small region called window. After each window is completed the next window is traced. However the efficiency can be improved by computing the forward metrics of the next window while the backward metrics of the previous window is being computed. This thereby reduces the latency of whole trellis from a frame size to a single window size. However performance loss is encountered while computing the metrics in each window. Therefore a small window called learning period or guard window is allowed to compute the dummy backward metrics in order to achieve more accuracy and minimize the backward metric errors. Figure 3.1 illustrates the implementation of sliding window algorithm (7).

Assume code length of N, the window length being W and the learning period being L. The algorithm divides the entire trellis into windows of size $W + L$. The forward metrics are computed first till the end of entire window $W + L - 1$. Then backward metrics are initialized with the last computed forward metric and dummy metrics are computed for entire guard window of L. Then actual backward metrics are computed for

So, he has got this one forward recursion if we start at 0 and go up to w minus 1, which is actually a window with, but you do not stop there, you go all the way in the forward recursion up to w plus l minus 1. So, there is a extra l that you have to go and then start your reverse recursion from w plus l minus 1 all the way down, but you make decisions only from w minus 1, you cannot make decisions from there.

So, the same thing you do and then you continue with the second forward recursion from where you left off w plus l minus 1 and then the backward recursion will go all the way down to w minus 1 and then you start the computation. So, you have to overlap your windows like this, it is just a small trick, but you should do it carefully the crucial parameters are w and l what w you pick, what l you pick. All these things you have to optimise in your simulation. So, what do you do, how do you pick w and l, there is no theoretical formula. So, you first pick different w s and you keep plotting and you see which gives you the good performance and he has done whole bunch of that.

(Refer Slide Time: 31:15)



in order to make design simpler and consume less memory area and power. However such techniques result in the performance loss in terms of BER and FER. Therefore it becomes essential to study the effects of such algorithms. Figure 3.2 shows the BER performance of Log MAP, Max Log MAP and Sliding Window Max Log MAP turbo decoders in AWGN channel. The simulations are carried on 3G standard turbo encoder using BPSK

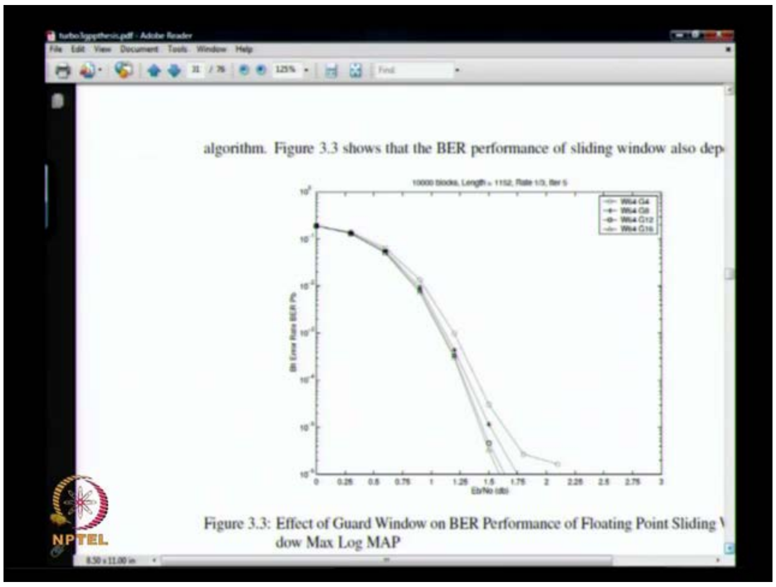
Figure 3.2: BER Performance of Floating Point Log MAP, Max Log MAP and Sliding Window Max Log MAP

modulation. The number of iterations is kept as 5 and the frame length chosen is 1152 and rate $\frac{1}{3}$ and number of blocks simulated are 10000. The window sizes are also varied from 32 + 8 to 128 + 16. It is observed that Max Log MAP is a close approximation of Log MAP with a performance of loss of 0.4dB. Increasing the window size produces good approximate of sliding window Max Log MAP mapping to the Max Log MAP decoding

I will show you how what all that he has done. The first thing is this, I think the hold and graph for some reason is not working. So, the first plot he has done is that and then there is a sliding window that he has done. He has not put down here what are the what is w and l, but I think he has optimised it later on in this other plots for that. So, the first thing I want to point out is that what e b o r n naught is achieving 10 power minus 6.

Look at that 1.1 encoded is at 10.5 at what kind of gain is that 9.4 d b coding, it is huge of course the rate is 1 by 3. Maybe you do not like it too much, but still the coding gain is huge in e b o r n naught. Of course, rate is being accounted for in e b o r n, so that is how the turbo codes work. So, they are really, really, really very nice. So, let us start looking at the other things what else do we do?

(Refer Slide Time: 32:24)



algorithm. Figure 3.3 shows that the BER performance of sliding window also dep

Figure 3.3: Effect of Guard Window on BER Performance of Floating Point Sliding W
dow Max Log MAP

One third is maybe less than 0, it is about minus something minus 0.5 maybe. So, here he has looked at effect of what he calls is guard window, I think guard window must be the l. How large is l for a fixed w, w is fixed at 64 for some reason and then he is looking at g 4, g 8, g 12, g 16. So, l is 4 8 12 and 16 and you plot different things as you increase l. Of course, as you increase l the performance becomes better and better. At some point it will start bunching, so you see the bunching is happening at 12 and 16.

So, you pick something like 16 as you are the lag or the what he is calling the guard window. The guard that you have to do to make sure you get good results. So, this is what you do, this is how you design if you go to a company or something and you have to implement a turbo code, this is what you will be doing. First you write a Mat lab program and then you test all these parameters. So, you simply keep simulating it different parameters plot this curves keep on you have to plot and pick the different parameters to give you a good performance. So, this is for effective guard window and then quantisation. So, the quantisation is the next part.

(Refer Slide Time: 33:36)

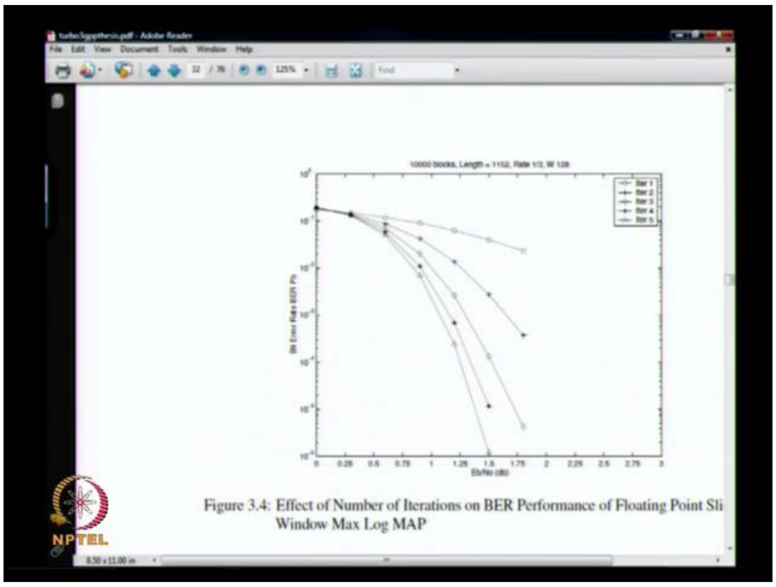


Figure 3.4: Effect of Number of Iterations on BER Performance of Floating Point Sli Window Max Log MAP

Before that he has done number of iterations. So, how do you stop how do you know when to stop, how many iterations do you do. So, you plot probability of the first iteration. So, this is also I think points out this also nicely brings out the effect of the turbo principle. The first iteration look at the curve the topmost curve, it is not doing anything by the time.

You do 5 iterations what has happened it is taken a drastic fall. Look at the gain from the first iteration to the fifth iteration, it nicely brings out the turbo effect, I mean how the iterations really improve the LLRs. So, we made a decision to stop at 5 iterations after plot like this sixth iteration will give you very-very small benefit, it will start bunching. So, at that point you pick your number of iterations as 5 and this is still floating point and I think then, he did some histograms. I think I do not want to go into details and this is this is fixed point.

(Refer Slide Time: 34:42)

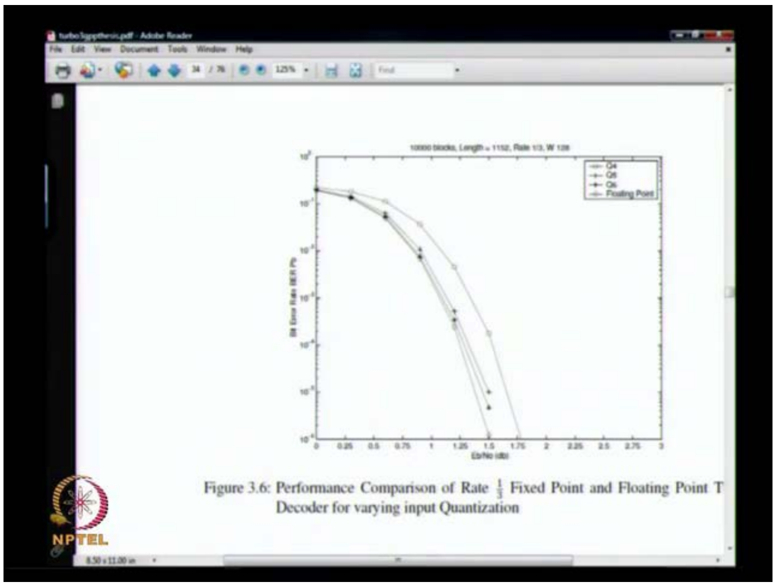


Figure 3.6: Performance Comparison of Rate $\frac{1}{3}$ Fixed Point and Floating Point T Decoder for varying input Quantization

So, he has got q 4, q 5, q 6 I do not know what it means really, but three different ways of quantising the received values maybe one is 4 I think. It is not four bits it is maybe there is a common 3 part and then it is either 4 bits on top of that and then another 5 bits or another 6 bits something like that. So, three different quantization and then floating point the floating point just about the best you can do in computation. No anybody has found another way of representing real numbers or something like that to beat floating point.

So, floating point is really good, so beating floating point with finite quantisation is hard you have to. So, really a lot of work if you want to beat floating point, so floating point you can take as the standard. So, that is the left most curve here then, what you, you slowly increase your quantisation number of levels that you are using to make it in infinite precision till you get sufficiently close performance to floating point.

And that point you can stop, so he has found that q 6 for instance is very-very close to the floating point line. So, you pick that q 6 as your quantisation level and he has used w is 128 and rate is one-third etcetera. So, this is how you pick each and every one of your parameters in your decoder. All these things are important in practise. So, if you are implementing a turbo code you have to worry about how many bits are you going to get per symbol?

You will never get floating point, so after all there some analog to digital converters sitting there. If it converts the samples into a digital data for you it cannot be more than 8 bits it is very hard to imagine anything more than 8 bits. So, 8 bits is a good number to think for that quantisation, how you implement your turbo decoder, how does it work all that you have to study. So, each and every one of these aspects you can study and again you have to go back to the simulations.

Unfortunately it is not the code which has great structure where you can just analyse and just get some data out. You cannot decide number of quantisation in table without doing simulations. It is very-very difficult to do that.

(Refer Slide Time: 36:45)

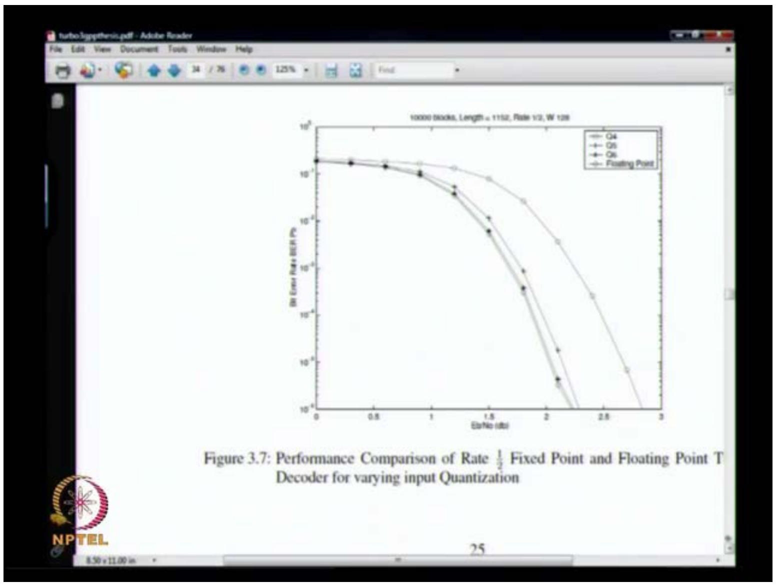


Figure 3.7: Performance Comparison of Rate $\frac{1}{2}$ Fixed Point and Floating Point T
Decoder for varying input Quantization

25

What is this, this is for rate half. Again rate half the 3 g p p will specify how to go from rate one-third to rate half. I believe it is even an odd functioning when you do that, you go to rate half and you see there is it is not as good as rate 1 by 3, the coding gain is dropped, but then you gain in rate you go to rate half. So, but, still there is significant coding gain you get 10 power minus 6 bit error rate at around 2.2 2.3 e b o r n naught that is still about 8 d b of coding gain at rate half, which is a really good.

If you remember the best convolutional code plots they were at around 5 d b 6 d b. So, there is still like this huge 3 d b gap from the regular convolutional codes that you are using at rate half. And with again q 6 quantisation as he calls it I do not know exactly what q 6 is, but q 6 quantisation is able to get very close to floating point so you pick that as the that is your point. So, this is rate two-thirds this is also specified in the 3 g p p standard. Once again you do similar curves and you see the there is some kind of a flooring effect that you can kind of see at rate two-thirds.

(Refer Slide Time: 37:56)

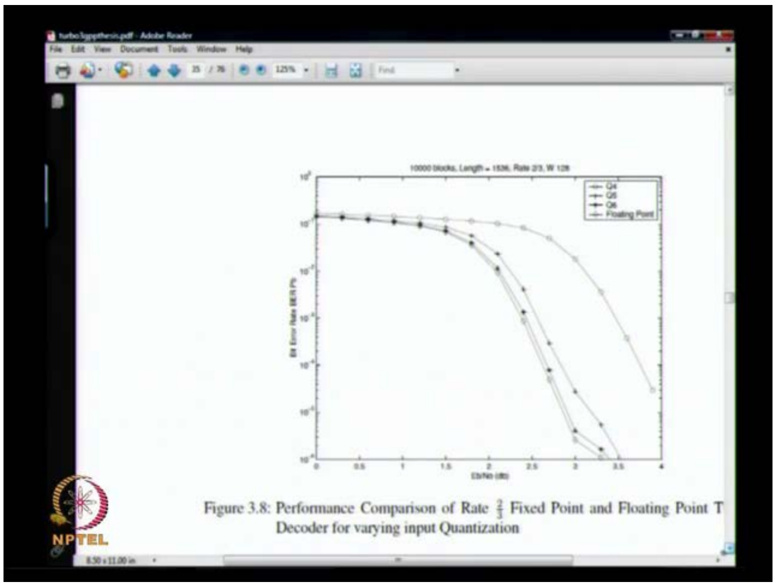


Figure 3.8: Performance Comparison of Rate $\frac{2}{3}$ Fixed Point and Floating Point T
Decoder for varying input Quantization

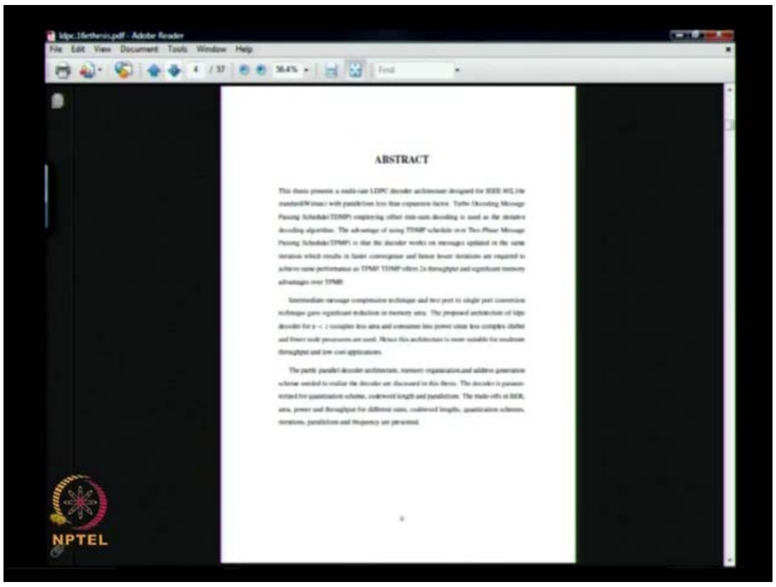
So, it does not this is what I mean when I say it is only for finite f e r. Of course, you cannot really avoid this floats. So, you get this and that is about at 3.3 d b or so for rate two-thirds again as a significant coding gain for and the higher rate. So, these are plots that you have to do. So, after that you can optimise power that is something that he did, but anyway the general idea I want to convey is, if you are implementing turbo codes in a company or for any other purpose, you will make a lot of plots like this.

So, you have to create plots like this to justify your design choices. What are the various design choices here, you have to decide about window length that is a very pretty practical thing to decide on how many bits you are going to put per symbol that is something you are going to decide. How many iterations you are going to do, all these things you have to decide based on these kind of simulations, there is no way around. A 3 g p p specifies the Intel lever it is the q p p Intel lever.

So, usually in the standards the Intel lever is specified. So, the standards will completely specify encoding what do they do not specify the decoding. So, that is where you have to do the receiver is what you have to build the transmitter is usually standard, that is what standardised. Of course, there are standard there will be decoders available; I mean implementation is available, but it is good to optimise.

So, this is for the turbo code I wanted to show you a similar thesis for LDPC codes. In fact both these guys worked together and they were kind of competing with each other to see which is better LDPC or turbo. So, that is the there is a kind of competition that we had and anyway both these guys did some good stuff. They were both VLSI M tech. students not communication M tech. students. So, I am saying that is why they did good work.

(Refer Slide Time: 39:58)


ldpc16ethesis.pdf - Adobe Reader
File Edit View Document Tools Window Help
NPTEL

ABSTRACT

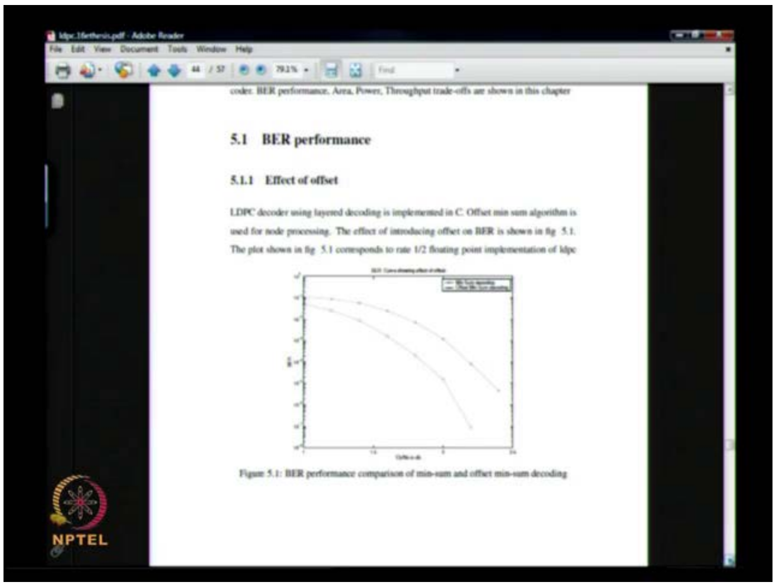
This thesis presents a multi-rate LDPC decoder architecture designed for IEEE 802.16e standard(Wimax) with parallelism less than expansion factor. Turbo Decoding Message Passing Schedule(TDMP) employing offset min-sum decoding is used as the iterative decoding algorithm. The advantage of using TDMP schedule over Two Phase Message Passing Schedule(TPMP) is that the decoder works on messages updated in the same iteration which results in faster convergence and hence lesser iterations are required to achieve same performance as TPMP. TDMP offers 2x throughput and significant memory advantages over TPMP.

Intermediate message compression technique and two port to single port conversion technique gave significant reduction in memory area. The proposed architecture of ldpc decoder for p < z occupies less area and consumes less power since less complex shifter and fewer node processors are used. Hence this architecture is more suitable for moderate throughput and low cost applications.

The partly parallel decoder architecture, memory organization and address generation scheme needed to realize the decoder are discussed in this thesis. The decoder is parameterized for quantization scheme, codeword length and parallelism. The trade-offs in BER, area, power and throughput for different rates, codeword lengths, quantization schemes, iterations, parallelism and frequency are presented.

Anyway both these guys I think Kiran is working with I cannot remember Mudassir also working. Both of them are working in Bangalore doing more VLSI stuff. So, let me see I mean where this was come this will probably much later maybe I can use this. When I want to it is not going very fast. So, maybe no it is not what I want. So, maybe it is here. So, this is what I want to show you. So, this is 8 o 2 dot 16 e, which is the y max standard, I believe the y max I mean dot 16 e.

(Refer Slide Time: 41:11)


coder. BER performance, Area, Power, Throughput trade-offs are shown in this chapter

5.1 BER performance

5.1.1 Effect of offset

LDPC decoder using layered decoding is implemented in C. Offset min sum algorithm is used for node processing. The effect of introducing offset on BER is shown in fig 5.1. The plot shown in fig 5.1 corresponds to rate 1/2 floating point implementation of ldpc

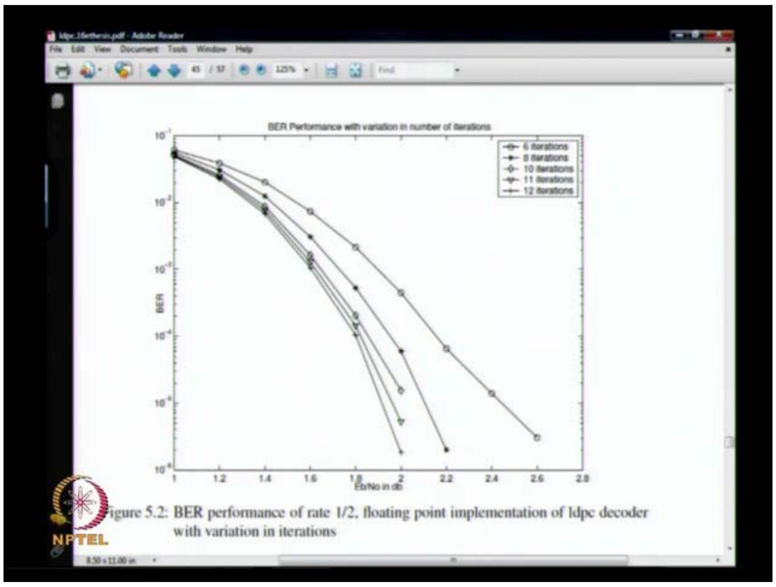
Figure 5.1: BER performance comparison of min-sum and offset min-sum decoding

You have the LDPC codes, but y max is chosen not to use it maybe others will use it. I think it is maybe even optional I cannot remember what is the exact situation is now. And Kiran implemented mean sum decoding; he did not implement the full scale 1. So, here is the first plot that you see this is the floating point simulation for a rate half LDPC code. I believe iterations 10 and then I think the block length is oh my goodness.

So, let us zoom in a little bit, so this is the floating point simulation for a rate half code. I think I mean I cannot remember the block length he does not seem to have specified anywhere. Maybe later on we will have those things also, but this is mean sum decoding and also the turbo simulations I showed were always max log map they were not log map.

And you can see similar picture you have there are two comparisons here. One is mean sum and the other is offset mean sum offset mean sum is slightly better and you can see about 2 d b you are getting 10 power minus 6 for rate half, which is what roughly the turbo code also did. So, for if you look at rate half turbo code LDPC are roughly the same in terms of performance. In fact the rate one-third you will never have LDPC codes, it has become very inefficient at that point it is better to do turbo if you want to do rate one-third. So, again once again effect of iterations how do you decide how many iterations to do 2 6 8 10 11 12 till you see the bunching up.

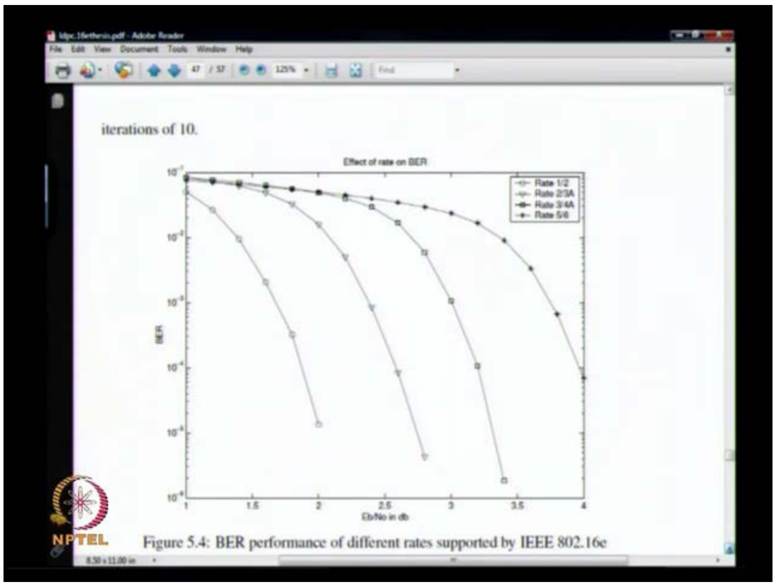
(Refer Slide Time: 43:11)



igure 5.2: BER performance of rate 1/2, floating point implementation of ldpc decoder with variation in iterations

This is all LDPC remember it is not turbo anymore doing LDPC and you do this simulations still you see the bunching up and you can see about 12 iterations seems like a good number to do and then effect of quantisation. Once again you do 4 bit quantisation 5 bit quantisation, 6 bit quantisation floating point see when it gets close and when you pick it effect of rate he called it effect of rate. So, anyway it does not matter it is again 2 3 0 4 I think the block length is fairly high 2 thousand or something 2300. You can see there are rate half rate, two-thirds rate, 3 by 4 and rate 5 by 6 codes.

(Refer Slide Time: 43:47)



iterations of 10.

Figure 5.4: BER performance of different rates supported by IEEE 802.16e

In the standard this is block length 2 3 0 4, 10 iterations he has done. You can see the curves is about 2 d b. Around 2 d b you are getting 10 power minus 5 here about 2 point i do not know 7 or 8 you are getting 10 power minus with the 2 by 3 for 3 by 4, it is just 3.4 or so and even 5 by 6 it is about 4 d b I mean it is 10 power minus 4. So, it is quite a good performance for LDPC codes. So, in fact if you compare the rate two-thirds code you will see the LDPC code is doing better than the turbo code for corresponding length.

So, it is just I mean for higher rates in general it is easier to design LDPC codes because you have to do. For turbo codes you have to look at a lower mother rate code, you have to puncture etcetera that turns out to be a little bit more sub optimal and this is something that you can do in LDPC. So, that is the plot and then this is code word length again this is again an interesting lesson.

(Refer Slide Time: 44:45)

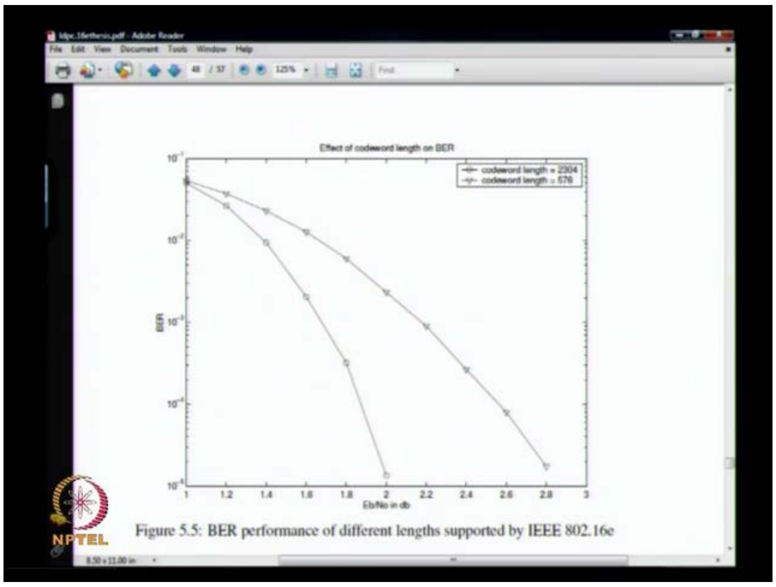


Figure 5.5: BER performance of different lengths supported by IEEE 802.16e

So, the same to same I mean the LDPC code 1 specified for code length of 576 and another is for 2304 same rate, rate half and look at the difference 2304 10 power minus 5 around 2 d b length 576, it goes all the way up to 2.8 d b. So, the longer block lengths are always better in turbo LDPC codes. So, longer you go it is always better 576, I believe is the smallest length specified in the y max dot sixteen d standard. So, I think that is I think then he talks about synthesis results and all that so.

So, what I am trying to convey by showing these two thesis, the main idea is if you are in the business of implementing these things in hardware, you have to worry about a lot more things. You have to worry about many things, but essentially the algorithm will not change it will be the same. So, you will have to make the decisions such as number of iterations, number of bits per symbol, and all these decisions are made using simulations that is all. You have your basic decoder working in 6 point then; you can make all these decisions. So, try anything you like and then pick the best so then you implement. So, that is what you do if you join industry today and building codes, this is the kind of work that you guys. So, I think that kind of concludes then hey what this is left most pardon.

Student: ((Refer Time: 46:38)).

Hey what why is this?

Student: ((Refer Time: 46:57)).

What do I do man any idea.

Student: ((Refer Time: 47:01)).

Alt tab.

Student: Start button.

Hey finally, you do it becomes an adventure. See this seems very well behaved with the pull, but that thing was the adobe reader was very bad.

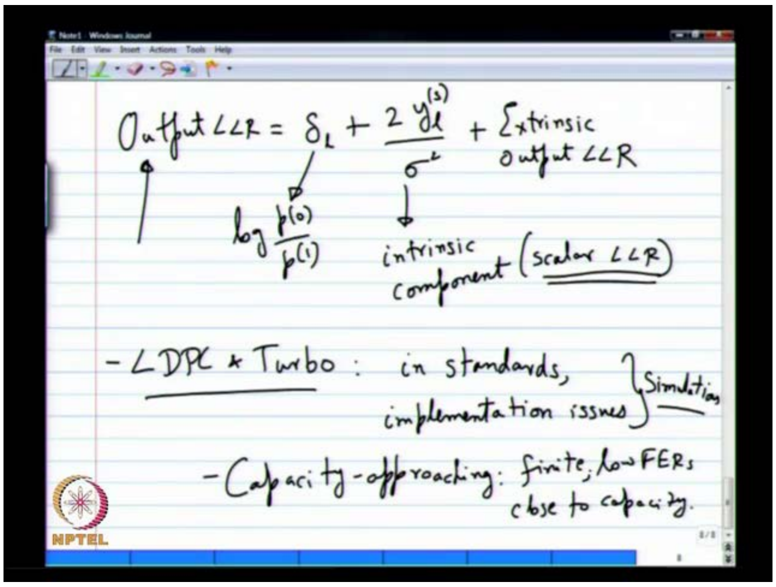
Student: So, when in general actually when the s n r is bad there will ((Refer Time: 47:52))

I think now the question about the standards, I suppose to the codes themselves. I mean I do not know what the standards allow; I do not if you can switch from LDPC to turbo or something, but usually it should be possible. You are rate one-third maybe is the lowest and beyond rate one-third also they do. In fact what they will do after one-third they will do repetitions so beyond one-third give up hope on code design, you simply do repetition keep on repeating anyway.

There is an absolute limit on e b o r n naught beyond, which you cannot do anything that is something like a minus some 1.69 or something. So, you cannot do anything below that. So, if it is below that you have to boost up something else. So, I mean one-third is like the lowest code rate you will see in standards, very few people go below one-third and LDPC is stops at one-half. Very few standards go below one-half, but I mean believe me 1 d b 2 d b and all is really-really low s n r. You are looking at very low I mean in e b o r n naught that is really low.

Most systems will work at around 3 4 at least. At least you will adjust your power budget that way; you would not bet on 1 d b, 1 d b will be like the extreme case when nothing else is everything else is failed it will succeed kind of situation.

(Refer Slide Time: 49:28)



$$\text{Output LLR} = \delta_l + \frac{2y_l^{(s)}}{\sigma^2} + \text{Extrinsic output LLR}$$

δ_l: $\log \frac{p(0)}{p(1)}$

$\frac{2y_l^{(s)}}{\sigma^2}$: intrinsic component (scalar LLR)

- LDPC & Turbo: in standards, implementation issues } Simulations

- Capacity-approaching: finite, low FERs close to capacity.

What I know about the wireless standards I mean, you have to ask somebody else who knows more about whether you can switch or not. So, let us few minutes few seconds left so, let us quickly summarise. So, LDPC and turbo is what we have looked at and there are some general principles, which I spoke about very briefly, but let us not we will get into that in the next lecture. Maybe in general these are in standards, they will continue to be in standards and there are lots of implementations shows, which are interesting, but people still work on.

So, implementation issues that people still work on. And most of these choices are done based on simulations. Simulations play an important role and that can be a nice theory. I do not know that is the first question I mean for implementation at least nobody bothers with the theory if you can do it is fine. And the one thing to remember is the usually when you meet people everybody talks about LDPC and turbo capacity approaching. So, you should know in what sense these are capacity approaching, what is the sense in which the capacity approaching.

They can achieve finite the important word is finite low f e r s close to capacity. So, you should remember first thing you should remember is as long as you stick to finite block lengths, you can only achieve finite error rates you cannot go below finite. So, the issue here is suppose, if I let my block length go to infinity, can I still get arbitrarily low f e r s? It turns out not all LDPC codes do that I mean you cannot show a capacity of achieving what there is a gap to the capacity at which today people have been able to show that you can get arbitrarily low frame error rates for LDPC codes.

Similar thing is true for turbo codes and turbo codes in fact much lesser theory is available. LDPC is a little bit better in terms of theory, but the state of the art today is you cannot show low arbitrarily low block error probabilities for arbitrarily long block lengths and getting to capacity. If you are willing to be a little bit away from capacity not too far little bit away then you can show arbitrarily low frame error rates for LDPC codes. So, that is the kind of situation that people are in. Of course, deterministic constructions are still open so far people do is an random looking Intel levers even for LDPC. It is random computer search based design deterministic constructions are still unknown for these things.