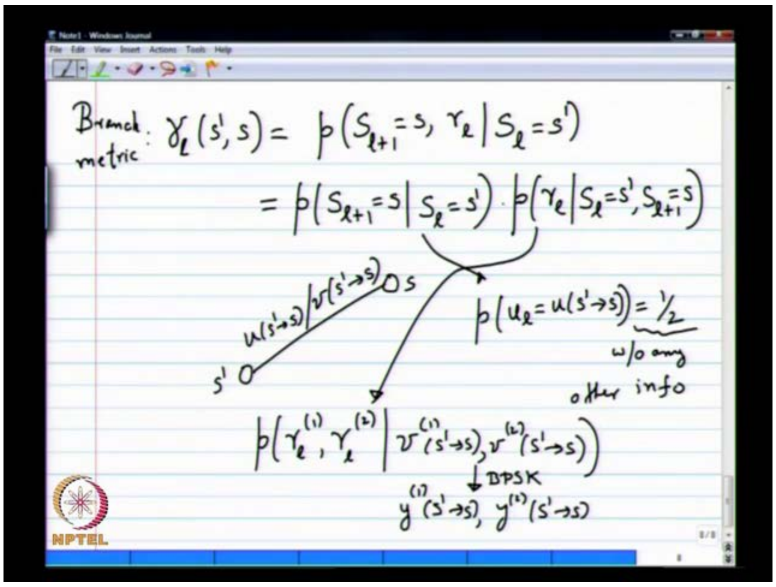
Coding Theory
Prof. Dr. Andrew Thangaraj
Department of electronics and Communication Engineering
Indian Institute of Technology, Madras

Lecture - 35
BCJR and Max-Log-MAP Decoder, Introduction to Turbo Codes

(Refer Slide Time: 00:14)



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

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

$p(u_l = u(s' \to s)) = \underbrace{1/2}$ w/o any other info

$p(r_l^{(1)}, r_l^{(2)} \mid v^{(1)}(s' \to s), v^{(2)}(s' \to s))$

$\downarrow$ BPSK

$y^{(1)}(s' \to s), y^{(2)}(s' \to s)$

Okay so at the end of the last lecture, we were talking about the branch metric. So, let us see real quick how to compute this. How do you… What is the first step we can do? Let us see who is going to tell me the very first step. What can we do? Any ideas? Condition Sl plus 1. So that seems like an interesting suggestion. So, write it as probability of Sl plus 1 equals S. Given, Sl equals S prime, times probability of Rl given. Is this what you wanted to say? Condition on Sl plus 1, Sl equals S prime comma S l plus 1 equals S. It seems like a natural good thing to try right? So, no, is this correct? The question is correct. What about this guy? What is S comma S prime? Remember?

So, yeah so it is just a trellis. Always think in terms of the trellis. So, you are thinking of S prime here and S here and then asking, what is the probability that S l plus 1 equals S given that S was l was S prime? What is this probability? It is going to be 1 by 2 right? It is the probability of whether the input was 0 or 1. So, there was only 1 bit input here. So, it could be either 0 or 1. So, it is basically the probability of U l. So this is going to be Ul slash some output right? Corresponding output?

So, I will write down the output soon enough, but Ul is this. So, this term is basically probability that Ul equals what? Whatever that should be right, right? So, remember what happened? The input here is a random variable Ul. So, that Ul has to correspond to whatever was on that branch. So, I do not know. I mean, we can have some notation for it. What notation can we have? So, let me say U of S prime to S. So, that is all. So, that is the input corresponding to the branch S prime to S. It could be either 0 or 1. So, this is this is A priori probability. So, this is going to be half. Half without any other information.

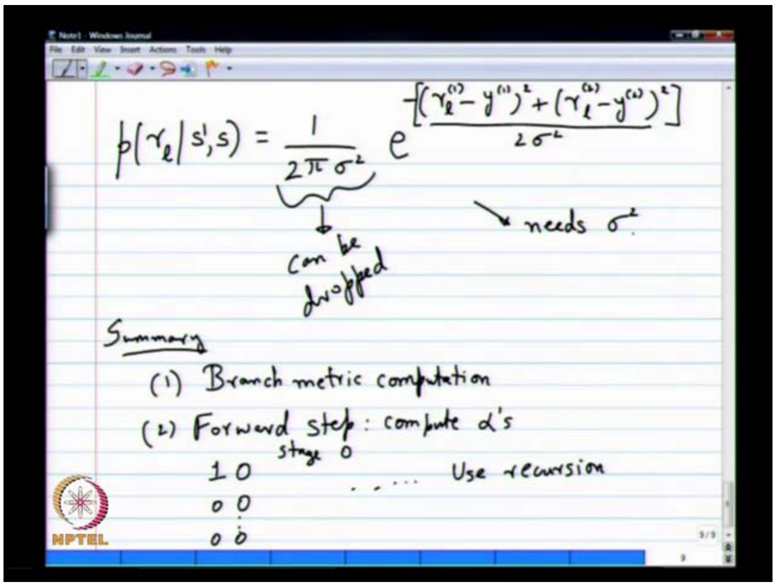
So, keep this in mind. This term we will revisit later when we talk about turbo codes. When we decode, there might be something here. There might be some other information you might get on this. If you do not have any other information A priori, you have to say it is half. Where nothing else we can do about it usually it is going to be half. What about this guy? Yeah. So, it is going to be some normal distribution right? See remember S prime to S is some branch. There was an input and there will be an output. So, what is the input and the output? We are going to… Let us write that down.

We are going to say this input is U of S prime to S and we can say that the output will be… Yeah, some I think the notation I have used is what? What notation have I used? V right? V of S prime 2. So, that is the output. So, this V will correspond after b p s k encoding. It will correspond to either some plus 1, minus 1, minus 1 plus or something. So that those were the 2 transmitted symbols. R l is basically Rl 1 and Rl 2. So, that is what? That is how you compute. This will be basically, probability of Rl 1 comma Rl 2 given what? Given V, V 1, S prime to S comma V 2, S prime to S right?

So, this output will have 2 bits. Those 2 bits are getting b p s k encoded and then sent on the channel to get Rl 1 and R l 2 and that is exactly this guy. How will you compute this? So, I will have a b p s k encoded version of this. So, the b p s k encoded version of this, the b p s k encoded version of this, we can use some notation. Is there any notation that we have used for the b p s k encoded version y? No y.

So, it is going to be y 1, S prime to S y 2, S prime to S. I will simply use y 1, y 2. It is not too confusing. Use y 1, y 2. Basically, Rl 1 is Gaussen distributed with mean y 1, S prime to S and Rl 2 is Gaussen distributed, with mean y 2, S prime to S. Only that variance is important and you can simply compute the… And these 2 are also independent right? So, you have a 2 bi variant Gaussen which is independent.

(Refer Slide Time: 06:26)



$$p(r_l / s', s) = \underbrace{\frac{1}{2\pi\sigma^2}}_{\substack{\downarrow \\ \text{can be} \\ \text{dropped}}} e^{-\left[\frac{(r_l^{(1)}-y^{(1)})^2+(r_l^{(2)}-y^{(2)})^2}{2\sigma^2}\right]}$$

↘ needs σ^2.

Summary

(1) Branch metric computation

(2) Forward step : compute α's

stage 0

1 0 . . . . . Use recursion

0 0

0 0

So, it is simply a product of each of those things. So, this works out very easily to the following. So, let me write that down finally. Probability of Rl, given S prime, S will basically be 1 by 2 pi sigma square, how did I get 2 pi sigma square? It is root 2 pi sigma squared. Then you will have e power minus what? R 1 minus R 1 l minus y 1 square plus R 2 l minus y 2 square by 2 sigma square. Remember this minus is for everything. That is the formula for this probability. Is it okay? So, the branch metric for b c j r is computed like this alright?

So, the difference between the b c j r and the viterbi algorithm in terms of the branch metric is, in the viterbi you do not need sigma for computing the branch metric. You do not need to know the channel noise variance. In the b c j r you have to know the channel noise variance. If you do not have sigma, you cannot compute this. So, one can argue that this is an inconsequential factor. Why is this inconsequential? It can be dropped because it will be there in every possible term that you compute and finally, when you compute the l l r it will cancel. You will do a numerator and the denominator and it will cancel finally, in the l l r. There is no problem. So, that can be dropped.

So, sigma square does not show up there, but it shows up inside the exponential and you cannot drop it or you cannot cancel it in any which way you want. So, it is going to show up in the computation. You need sigma square for computing the branch metric in the b c j r algorithms. Of course, this is not the branch metric by itself. You have to multiply this with half right? Half if you have no A priori information or if you have some A priori information that lets to be multiplied, so that is an observation. How do you compute sigma square? Not compute. How do you write word as estimate sigma square?

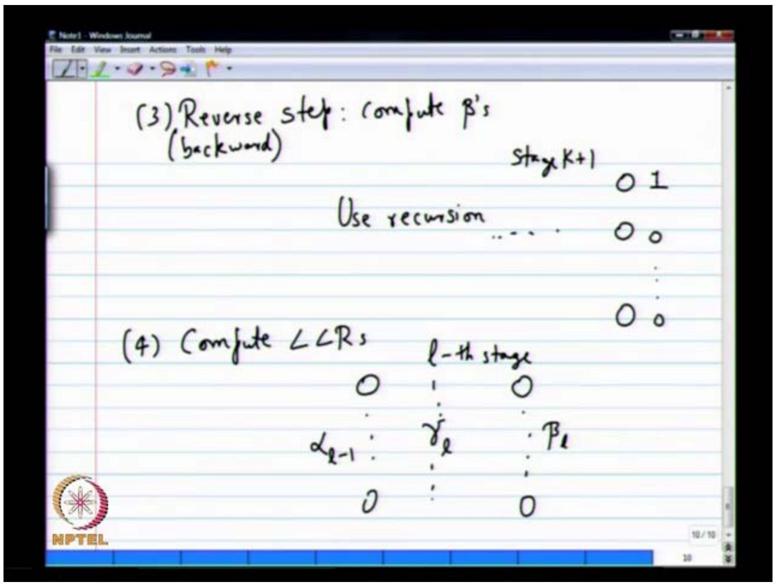
Yeah so, you have to estimate sigma square. So given your received values, you can estimate it in several ways. There are standard algorithms. If you are taking the, if you… did you take? If you took the estimation theory course that was offered, when was that offered? Last semester or something, you would have learnt about it. Estimating parameters is a standard thing. So, there are ways of doing it to any degree of accuracy. So, you can estimate sigma square. So, once you estimate it, you can plug it into your formulae.

So, usually how it is done in receivers today, is you look at your received values R 1, R 2 and so on and from those values itself you estimate your sigma square first. You do that with some reliability and then you use it once again in the computation. That is a very standard that is used in receiver implementations. The advantage to doing that is then your decoder block is independent of anything else that you have in the decoder in your receiver implementation. It might have so many other blocks, you might want to just independently estimate sigma square in your own way inside your decoder and implement it.

That is one way of doing it. There could be other ways which are more interesting and for people to look at, but this is one way of doing. Alright, so let me quickly summarize how this is going to work out in the l th stage. How the b c j r algorithm is going to work out? You are going to first have the forward step. The first step is the forward recursion. The first step before all of that is branch metric computation. That is the first step. So, once you do that, the gammas for every single branch in your trellis, in your entire trellis has been computed. So, you do that first. The next step is forward step. What do you do in the forward step?

You first start at the zeroth stage. At stage zero, how will you start at stage zero? You put 1 here and you put everything else as 0. These are the alpha. So, forward step you compute alphas right? At stage zero you start with this, that you start your recursion and then in the l th stage what you do? You basically use the recursion and then so on use the recursion to compute all your alphas. So, you go through and compute all your alphas. Remember, when you terminate the alpha, computation will slightly change. You will compute only for the sub part of the trellis which is really retained in your termination ok?

(Refer Slide Time: 11:50)


(3) Reverse step: compute β's (backward)

Stage K+1

Use recursion .....

$0 \quad 1$

$0 \quad 0$

$0 \quad 0$

(4) Compute LLRs

l-th stage

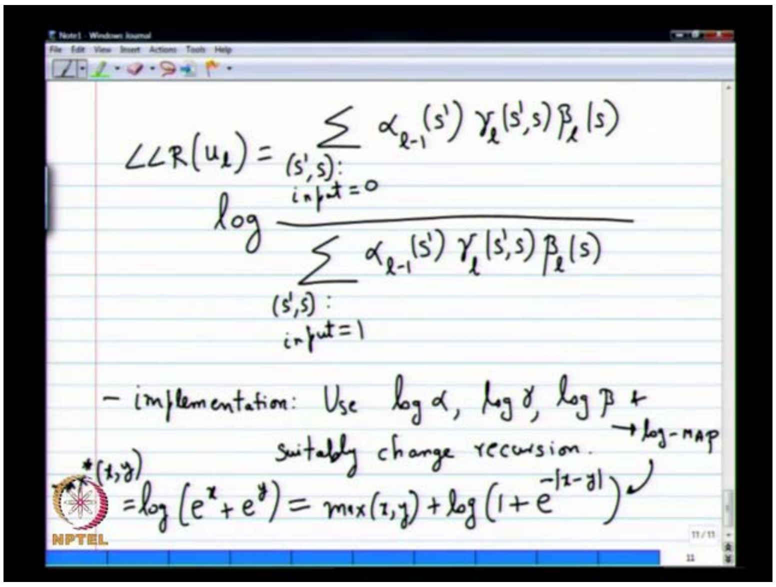
$\alpha_{l-1} \quad \gamma_l \quad \beta_l$

You can do that and then what do you do is the backward step or let us call it the reverse step. It is also called the backward step. At the last stage, stage k plus 1, on the right of it what do you do? You put 1 for the beta. So, this is for computing betas and then you put zeros here for the remaining stages and then you use recursion alright?

Alright, so after the branch metric say being computed, you can do the forward recursion. You can also do the backward recursion. Once you have finished branch metric computation, you know the gammas. Once you have finished forward steps, you know the alphas. Once you have finished the backward steps, you know the betas. So, what can you do now? Finally, compute LL r s. So, I will show you how to compute the LL r for the l th stage.

The l th stage you have a bunch of stages. You have the alpha l minus 1 on this side and then you have the beta l on this side and you have the gamma l in the middle branch metrics. The branch metrics are in the middle. The forward state occupation probabilities are to the left and the backward state occupation probabilities are to the right. How do you compute LL r for u l the input in the l th stage?

(Refer Slide Time: 13:25)



$$LLR(u_l) = \log \frac{\displaystyle\sum_{\substack{(s',s):\\ \text{input}=0}} \alpha_{l-1}(s')\, \gamma_l(s',s)\, \beta_l(s)}{\displaystyle\sum_{\substack{(s',s):\\ \text{input}=1}} \alpha_{l-1}(s')\, \gamma_l(s',s)\, \beta_l(s)}$$

– implementation: Use $\log \alpha$, $\log \gamma$, $\log \beta$ + suitably change recursion. $\rightarrow$ log-MAP

$${}^{*}(x,y) = \log(e^x + e^y) = \max(x,y) + \log\left(1 + e^{-|x-y|}\right)$$

You have to have a summation in the numerator over all s prime s, such that input equals 0 and what will you multiply? Here, you have s prime gamma l, s prime comma s, beta l of s and then what will you do here? Of course, I forgot the log. The log is all important s prime s, such that input equals 1 and then you do the same expression. That is it. Now, you have soft information on each of your message bits. In this final expression, it is also clear why any arbitrary scaling of gammas does not matter right? I had the 1 by 2 pi sigma square right? If I had some scaling for every gamma, then clearly in the ration, it is going to cancel.

So, in the LLr computation, the constant outside factor does not play any role. So, I can drop the 1 by 2 pi sigma square. The exponential obviously I cannot drop, so it will play an important role. Any questions on b c j r? So, given if you have enough experience coding in mat lab or c or any language it should be quite trivial for you to write a program for b c j r. It is not so difficult. At least you can write it down. What problems do you think you will face if you write it on a computer which obviously has a limited precision? What problem will you have? Yeah, there will be problems in over flow, under flow, etcetera.

So, for instance as your k increases, what do you think will happen to these alphas and betas? Gamma is going to be fairly well behaved. So, there is no problem. What will happen to the alphas and betas for large k? Remember, these are probabilities and there is a comma. There is a joint probability for a long vector and r is actually a huge vector. So, everything is going to go down in probability and value. Everything is going to keep on dropping and eventually values will become so low that you cannot rely on any computation.

So, it is very common for computation purposes to not implement this b c j r algorithm in the probability domain with alpha beta gammas as a probability, but usually people implemented and the log of probability domain. So, you simply take log of everything. You take log of everything, so instead of alpha you keep log alpha, but then what will happen to the recursion? There is a problem there. Look at the recursion. How do you deal with the recursion?

Ok you have a summation. Summation is not very well behaved with logarithm so that will be a problem, but you deal with it. You do that computation. It can be done in a smart way if you like. There is a smart way to do this recursion summation of products. When you take logarithm, when everything is log, so you put log instead of gamma also, take log of that also. You have log of sum of exponentials. There are easy ways of implementing that, mean v l s i. That is a way in which you can implement it in a smart way for instance. Whereas there are interesting ways of doing that, I am not going to talk about in detail here, but that is how it is usually implemented.

When you write a program, you keep log of everything. When you do that, then your precision does not matter right? Log is quite, it covers a wide range with good accuracy. So you do not mind at all. So, that is something that has done. So, that is something to keep in mind. So, in implementation use log alpha, log gamma, log beta and suitably change recursion. So, in that 1 operation that you have to do, is log of summation of exponentials right? Something like this, you have to do log of e power x plus e power y right? Am I right?

So, this is the expression that you have to compute again and again. When you do it in recursion, when you do it in logarithm domain, do you see that? When you do the recursion, it is alpha summation of 2 products. You have to do log of e power x plus e power y. Maybe not just e power x power y, but several other terms, but that is just a repetition of this. Once you do this, everything else is also the same. Do you see that? So, this is the crucial operation. There is a way to write this. So, this is usually denoted as so you can show an interesting property here. So, let me just quickly look it up and write down what this property is.

It is not difficult to prove. You can show this is basically max of x comma y plus log of 1 plus e power minus mod x minus y. You can show this. So, this is quite an easy relationship to show. Try to prove this. So, for this reason this operation is usually denoted max star max with a star on top. This reason is it is related to the max, but with an alteration, but how large will this value be? What is the largest possible value for log of 1 plus e power minus mod x minus y? Log 2 right? So, what is log 2? Well base e m taking about base e. So, it is not very large. It is less than 1 definitely right? So, is it less than one? Yeah, it is less than 1. Say point 6 7 something.

So, it is almost max except for the small alteration right? Do you see that 1 plus e power minus mod something right? So, it cannot really go very large. Maximum value for this is 2, right; 1 plus 1 2 log 2 base e also very small so it cannot be a very large number. So it is almost like max except for this non-linear term that is adjusted. So, that is why I said going to the log domain is not a big deal. Even in v l s i, all you have to do is implement this log lookup and that too it is a very small function and you can do it very easily. So, you do that, and then you get this approximation. So, what people do further in implementation is what? What is the most obvious thing you can do here?

Simply drop that extra addition. Just throw that extra addition term away. Who cares about that? It is anyway less than 1. Simply keep it as max. Now, there are several advantages if you simply keep it as max. What is the first advantage? Computation is obviously simpler, but what is… What else can you think of as another advantage? (( )) No, actually 1 of the problems I pointed out here in the branch metric computation goes away if you replace it with max. So, what happens when you take log of this? Remember, I am taking log right? When I take log of this by branch metric will simply be some square distance divided by 2 sigma square.

If I take max, the division by 2 sigma square is irrelevant. So, I can throw this sigma square term out and I do not even need that. So, that is another advantage in not doing that extra non-linear term and only doing max, which max does not change with non-linear constant scaling positive scale. So, as long as you do that, max is enough. So, you throw away this non-linear term and that version of the decoder is called max log map. So, this entire thing is called log map and it is called log m a p. If you do the log implementation, it is called log m a p. For that you use max star. If you drop this extra, additional term and only use max, that is called max log map.

So, I think most people who implement b c j r on actual hardware use max log map. Very few people use log max map. There are huge advantages to it. There is of course, a performance penalty. It is not as good as the log map. There will be some penalty, but there are so many advantages that you live with it and there are approximations. You know what people will do is, they will not just take max. They will do a minor adjustment. There will be an arbitrary adjustment that is done to max to make it little bit better. So, you can really make that gap go away by doing this add of things and making it really, really good. There are papers, several papers that are written on max log map.

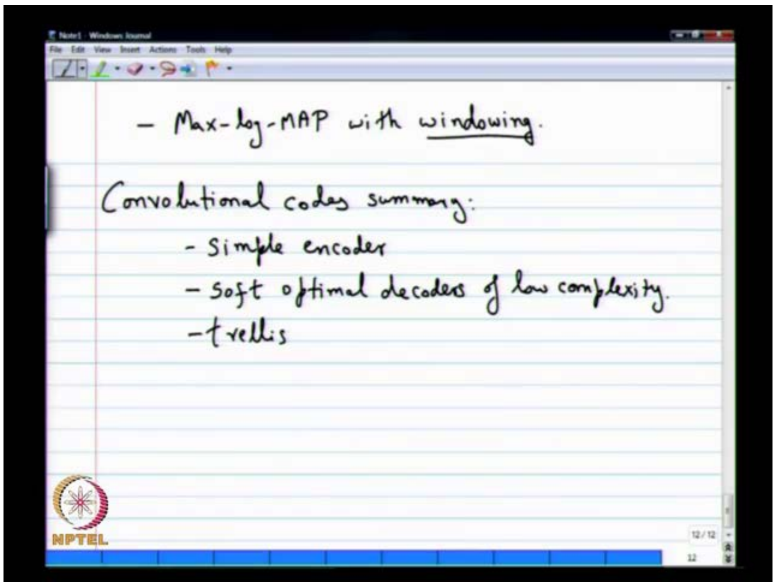
So, you can look it up and you will know what to implement. So, at least implementations that we have done in the department use max log map. It makes a lot of sense, works very well. So, that is I think all I wanted to say about implementation b c j r etcetera. Any questions? Like I said I mean you have to write a program to implement these things. You can write a very short mat lab code. It will not even take more than a few lines. It is just some, 1 loop for the forward recursion and 1 loop for the backward recursion and you should not be computing gamma first for everything.

So, when you implement of course, I wrote it down like this, you are not going to compute gamma everywhere and then store it. You compute it as you and when you need it for the computation. Stage by stage you can go. Another problem in implementation is if your k is very large, if you have thousand or something, then you have to remember too many alphas. The memory you need in your implementation will be very, very large. So, how do you overcome that? Well no, alphas you cannot compute as and when you need it. You need at least one direction full computed sand stored and betas you can compute as and when you need it, but alphas you cannot.

One way or the other you can only compute one thing. So, what is… We have seen this before with viterbi also. How do you overcome this long storage requirement? Yeah, you do something called windowing. So, windowing is the crucial idea. So, you start for, at a point go up to a certain point and then simply start your backward recursion there, but how can I start my backward recursion anywhere I want? I mean because I cannot do an initialization no?

What can I do? Yeah, assume equal probability. Go large, long enough and then assume equal probability. Come back in your backward recursion for a few stages without really using that data and then after a while your betas will become good enough and then you can use it. So, the same windowing idea is used in implementation. So, almost all implementations use like I said max log map with windowing ok?

(Refer Slide Time: 24:54)



– Max-log-MAP with windowing.

Convolutional codes summary:

- simple encoder
- soft optimal decoders of low complexity.
- trellis

So, this is a good compromise and it works quite well. Windowing is to make sure you do not have to store a huge number of alphas as you go through the trellis in 1 direction. So, that is something. It has to be carefully done and you cannot just stop anywhere because you have to assign some betas carefully. So, you stop a little bit further ahead, then comeback and then go ahead and do that. So, the question is comparing m l and m a p; m l and m a p. If you actually compare and plot, they will look pretty much identical. They will be on top of each other, at least in whatever that I have done. I have seen with convulsion codes, it is on top of each other.

There will be really no big change, but clearly the optimality is different. m a p is optimal for bit error probability. m l is optimal for block error probability. Remember, the m a p decoder is not constrained to output any valid path on the trellis. It will not output the valid path. It need not necessarily output a valid path, but it will you know. I mean you know so you have to be careful. When I talk about the trellis here you are directly outputting a message vector only. So, if you do that then of course, it will be a valid path, but if you also compute the parities and do m a p for that it may not be in the path. So, that is 1 thing, but in the b c j r clearly you are computing only message.

When you are computing only message, then it is not a problem. You will be getting valid paths. That is not a big deal, but in general the m a p will not necessarily give you the best path on the trellis. It will only give you the best set of bits, yeah for convolutional codes themselves. Like I said nobody does b c j r for convolutional codes. If they are part of a bigger construction, then you will see the advantage. I will talk about it. In turbo codes it makes a big difference. When you need soft decisions you have to do b c j r. Yeah, you will need LLr s for the message bits not just what it is. You want also, some LLr s.

Then at that point you need b c j r because viterbi will not give you LL r s. Any other questions? What do you do if it is not b p s k? That is the question. So, the formulas essentially are same. So, if you look at the way I derived it, up to the alpha beta gamma, I never used any b p s k. Only in the final expression for the branch metric gamma, I used b p s k. So, you can go back and think about that. Up to this formula there is no use of b p s k. If you are doing q a m and all that, first of all how do you do convolutional codes to q i m? You will have some mapping right? Some bits will get mapped to the symbols. That will play a role in these probability computations. That is all.

So, then up to these computations everything is valid. You have to change your branch metric computation depending on your mapping in the q a m and other things and there are ways of doing it. People have studied that and max log map approximations are there for that; all kinds of approximations. So, very well studied subject. So, that is an important question because in going with 4 g etcetera, people are doing up to 64 q a m even in wireless. This was unheard of before, but people are doing all these huge constellations. So, you will be facing those kinds of problems if you are implementing these. So, it is good to know what to do in q a m.

Like I said everything else is valid, only depends on the mark or chain properties. The only thing is, when you compute the branch metrics, you have to use your mapping. That is all. Depending on the mapping it will change. Maybe I will comment about q a m computations later on. So, I will talk about something else later on. How to do for something other than b p s k? If you do q a m? So, usually not, when people will do bit only. So, I will comment on that later. Also, usually the latest the modern way of doing modulation and coding is to make sure that they do not interfere with each other.

You know what I mean. You do the code only in binary and then you convert from binary to q a m. So, only these ok? So, if we are done with questions on b c j r, I want to just quickly summarize convolutional codes. What is the main selling point of convolutional codes? Why are they; simple encoder, that is the first selling point, very, very simple encoder. If you want to have a very, very cheap connecting code which is also good, then the encoder is simplest in convolution code. It is very hard to beat it. There is no way you can beat that and in addition what is the matching property of the decoder which is crucial? Low complexity, soft m l decoder, soft optimal decoders of low complexity.

So, these 2 are major, major selling points. It is a great, great combination and they provide good performance, but if you remember in some of the plots I showed you, convolutional codes by themselves after so much research and lots of work, they do not get close to capacity. There is always like a 3, 4, 3 d b gap at least. 3 or 4 d b gap from capacity. So, they do not get the capacity, but in practice, very few people try to get to capacity because capacity is like a 0 d b, 1 d b. So, that is really low s n r from any other things in your receiver chain. So the receiver chain is not just your encoder and decoder right? What else?

You must be studying communication. What else is there in your receiver chain? What is very crucial? What else can be crucial? You have to recover carrier. You have to recover timing. So, all those things, nobody knows whether they will work at 0 d b, 1 d b. So, for a long time they were also not so robust that they can work at 0 d b and 1 d b and all that. So, many other things are there. First of all detect whether some communication is happening. Where are your frames? Where you begin? Where you end? So, many other things are involved. Lots of single processing is involved before you can start your decoder.

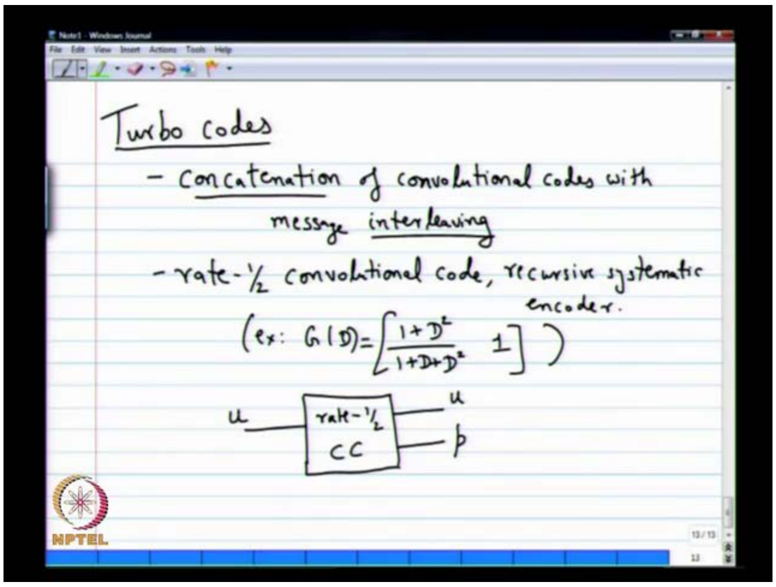
So, just because a decoder works at 0 d b, does not mean everything else will work. So, those were the problems. So, because of that, people when they do not want to stretch everything in a single processing to its limits simply work with convolution codes and it works very well. No problem in that and the trellis plays an important role. Trellis is a crucial thing that enables everything in a way. So, that is it. So, what we are going to see next is turbo codes. So, of course, they are quite storied today. So, everybody is at least showing some hint of a smile when I mention turbo codes. You are happy to see that we are talking about turbo codes.

So, like I said, for a long time people are talking, thinking the convolutional codes or any other codes that you can have will not really get to capacity. There was this talk of something called cutoff rate, computational cutoff rate. They are saying there is some other capacity which is not really capacity, little farther away from capacity which is the best you can do. Then in 93 or 94 I think; when did the paper come out? I think, I do not remember. Maybe 92, early nineties, there was this paper, from at that time; a little known French group I should say. So, it was from France and it had authors who were from, I think there was 1 author from Thailand and 3 others were French.

So, it was a collaboration, clot barrow was the main guy as it turned out later. So, it got published and they said they showed performance very close to capacity, 1 d b, 1 and a half d b and all that. For a long time people were shocked. Nobody understood what was going on and then the whole revolution started. So, today going to capacity is not considered a serious; I mean it is actually considered a serious requisite.

You know if you are building a communication system, everybody asks how far you are from capacity today. So, it has come to that level now. All of that started with this turbo codes. So, obviously now our understanding of turbo codes is also better, but when they originally presented it, it may not have been in this way, but right now the understanding is the current one. That is what I am going to talk about. The idea is so simple, so elegant and interesting. Hopefully it appeals to you and you like it.

(Refer Slide Time: 34:02)



Turbo codes

- Concatenation of convolutional codes with message interleaving
- rate-½ convolutional code, recursive systematic encoder.

$$\left(\text{ex: } G(D) = \left[\frac{1+D^2}{1+D+D^2} \quad 1\right]\right)$$

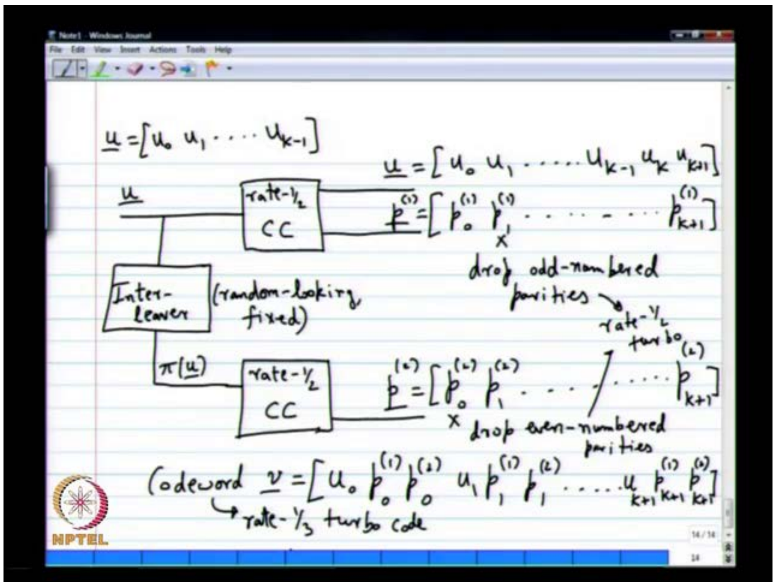
$$\begin{tikzcd} u \arrow[r] & \boxed{\text{rate-}\tfrac{1}{2}\ \text{CC}} \arrow[r, "u"] \arrow[r, shift right=2, "p"'] & {} \end{tikzcd}$$

So, turbo codes are what we are going to see next. So, the main idea here is concatenation. That is the first main idea of convolutional codes, with message inter leaving. So these 2 words are key words here. So, you do concatenation and then you interleave messages. So, I will be describing what is called parallel concatenation. There is also something called serial concatenation, but parallel is what I will be describing because that is what is used today. That has turned out to be good in many ways. Of course, that does not mean the other concatenations are not good, but parallel concatenation is accepted today as something very nice.

So, once again I will start with the simple example of a turbo encoder. So, like convolutional encoders, we will not talk about turbo codes directly. We will talk about turbo encoders and then the codes that they produced will be the turbo codes. So, how does a turbo code look like? So, what I have said here, there are convolutional codes. Of course, there are convolutional codes. So, we will say we will use rate half convolutional codes and with a recursive systematic encoder. So, that is all so commonly used. So, this will be the convolutional code that we use. So, you can for an example keep that standard example in mind.

If you want an example you can think of g of d equals 1 plus d square by 1 plus d plus d square and 1. So, this is an encoder. I will basically denote this by a picture like this. So, rate half, I will say CC for convolutional code. I will put 1 input and then I will put 2 outputs. So, if this is u, this is going to be u and this will be a parity which I will denote as p. Is that alright? So, that is a systematic code; systematic encoder. So, if you have u, u is going to come out and then there is going to be a parity. So this is my convolutional code. Now, this is going to get what is known as parallel concatenated.

(Refer Slide Time: 37:10)



$\underline{u} = [u_0\ u_1 \cdots u_{k-1}]$
$\underline{u}$
rate-1/2 CC
$\underline{u} = [u_0\ u_1 \cdots u_{k-1}\ u_k\ u_{k+1}]$
$\underline{p}^{(1)} = [p_0^{(1)}\ p_1^{(1)} \cdots p_{k+1}^{(1)}]$
drop odd-numbered parities
rate-1/2 turbo
Inter-leaver
(random-looking fixed)
$\pi(\underline{u})$
rate-1/2 CC
$\underline{p}^{(2)} = [p_0^{(2)}\ p_1^{(2)} \cdots p_{k+1}^{(2)}]$
drop even-numbered parities
Codeword $\underline{v} = [u_0\ p_0^{(1)}\ p_0^{(2)}\ u_1\ p_1^{(1)}\ p_1^{(2)} \cdots u_{k+1}\ p_{k+1}^{(1)}\ p_{k+1}^{(2)}]$
rate-1/3 turbo code
NPTEL

Let me show you. So, suppose you have a message u which is u 0, u 1, all the way to u k minus 1, u is first going to go through a rate half convolutional code and you would get u here and then k minus 1. Yeah, so I have to write down u k plus u k and u k plus 1 right? Do I have to? Yeah, I have to because it is a recursive code and I have to put in 2 things which are; I cannot just say 0 here. It is something else. Then I get the first set of parities. I will call this as p 1. So, of course, there is only 1 parity per symbol, but they still call it p 1. So, it is p 1, 0, p 1, 1 all the way to p 1 k plus 1. So this is the… What comes next is the master strokes so to speak.

So what do you do is you want to concatenate. You want to do parallel concatenation, but before that you also want to inter leave. So, you take the u and then do an inter leaving. So, I will call it as an inter leaver. What does an inter leaver do? It just permutes the bits in u randomly and the inter leaver has to be sufficiently random. It cannot be very deterministic. It cannot be that it switches everything in a very deterministic way. It has to be very random. So, that is the first condition, but I am not saying that now you just simply say inter lever. You assume this to be random. Some kind of a random inter lever and then what do you do?

The inter leaved version; so usually it is very common to denote permutations by pi. I will say the inter leaved version is pi of u. Now, this will go through another rate half convolutional code. So, in general it can be another code, but it is very common today to use the same code. So, you use the same encoder as before. Now, this will put out 2 outputs right? One of them will simply be pi of u and that is already there. So, you can simply drop that and only take the parity which I will denote as p 2. So, you have p 2 0, p 2 1, and so on till p 2 k plus 1. Now, there is something which is a bit; somebody might ask, think of this I do not know if any of you is going to think about it.

Code construction is not. I just wanted to talk about the 0 termination here. So, you might be able to 0 terminate the first encoder. You may not be able to necessarily 0 terminate the second encoder right? So, you just do; maybe you 0 terminate, maybe you do not. Maybe you terminate with something or maybe you simply stop at k minus 1. You do not have to really go to k plus 1. Maybe you 0 terminate may be you do not. You may not be able to do both of that. So, there is a problem there right? You see the problem? See if I do not, if I 0 terminate, what will happen? I have to send 2 more bits. So, maybe you do that. There is some issue about 0 termination here right?

So, I said I am going to ignore the systematic form. If I ignore the systematic bit, then clearly the 0 termination is a problem. I have to send at least 2 bits from there. 0 terminated bits. If you decide to drop that also, then you may not be able to 0 terminate here. So, all these things are issues, but these are just minor issues that affect you in a very small way. So, usually usual practice is to not 0 terminate the second encoder is that okay? Normal practice is to not 0 terminate the second encoder, but to zero terminate the first encoder. Is that ok? So, you 0 terminate the first encoder and then your inter leaver will have length k plus 1, k plus 2.

The entire; all the bits will be inter leaved and sent through the second one and you produce whatever parities. It ends in whatever state. It does not matter. So, you do not 0 terminate the second encoder. That is the common practice, but it is also possible to 0 terminate the second 1 also. Which means in that case you have to send 2 more bits. The 0 termination bits for the second encoder explicitly and that will reduce your rate a little bit. It may not matter in the end alright? So, there is a 0 termination issue, but common practice is to 0 terminate the first encoder and not worry about what happens to the second encoder. It can end in some arbitrary state. It does not matter.

So, then what is the output? Final code word is if you want to write it as V, it is basically what? u 0, p 1 0, p 2 0, u 1, p 1 1, p 1 1, so on till u k plus 1, p 1 k plus 1. Is there a problem with what I wrote now? Yeah, it is fixed. It is a fixed random permutation. Random looking. So, let me say random looking fixed permutation. It should not be a simple formula. That is, you know simple linear or very easy formula, random looking fixed permutation. As it turns out, people use some quadratic formulas and that is good enough. It should not be some simple thing, like every; I put it in a row, in a matrix and then put it in row wise and shift out column wise. Things like that you should not do.

It cannot be a very regular fixed, regular kind of inter leaving. It should be a random enough inter leaving. So, you pick something at random. Yeah, yeah, so I am going to talk about that. So, clearly the overall rate is 1 by 3 and this is something that you might want to live with, but usually current practice is to not go to that rate one by three code. What people do is to puncture. So, the common way of puncturing is to say that I will drop this guy. I will drop this guy. You drop all the even numbered bits in here in p 2 and drop all the odd numbered bits in p 1. Then what will happen to the rate? You get half. So, you drop odd numbered parity bits. You drop even numbered parities.

So, this is the common practice, but even if you do not, you have a rate 1 by 3 turbo code. So, that is the basic idea. Usually you drop these things and you get a rate half turbo code. So, this will give you rate half turbo code. This is obviously a rate 1 by 3 turbo code. So, what is the; I mean how could you probably justify this construction? I mean what is the motivation behind doing this inter leaving and concatenation? Any ideas? Well it is. They will not be the same. I mean when you change the order the parity in the way they show up is completely different right? It will be some other kind of bit. What is the idea? Noise is random. So change the point where the noise gets corrected.

Ok alright so all these things are I mean, I do not know. It sounds interesting to me, but I mean, it is not. Of course, the decoder is important. So, you can do a decoding. It is important, but more than that from an encoder point of view, what is it about this which is interesting? So, remember 1 of the points I made when I talked about good codes is you really do not know good codes. Bad codes you can easily find, but good codes you do not know very well. So, how easy is it to find a low weight code word here? That is my question. How can you find a low weight code word? So, let us try and repeat the convolutional code ideas.

Suppose I put in a weight one input, clearly the first systematic recursive encoder itself will kill it. I mean it will give you a long weight code word out and you cannot put weight 1. How did we get low weight code words for recursive convolutional codes? We had to use weight 2 and not all weight 2 code words, weight 2 messages result in a low weight code word. Only a one-third of them or something, so you have to find; pick the gap between the 2 very carefully. Only at some gaps you will get low weight code words. Now, what is the inter leaver doing?

Suppose I put in a low weight message, weight 2 message with the critical gap which will kill my first encoder and give out a very low weight code word, the inter lever is going to be a random thing. So, very high probability, what it is going to do? What is it going to do? It is going to make that gap bad for or good for the second recursive systematic encoder. So, I am going to get a long weight code word on that side. So, it might even be possible to actually fix an inter leaver which will always do that. Maybe it is not possible. I do not know. It might be possible.

So, at least it will be possible to come up with an inter leaver which will with very high probability always do that for weight 2 inputs. Then it has become much harder to find the weight 2 input which will give you a low weight code word right? Slowly as you go larger and larger weights of input, eventually your code word will any way become long and then you cannot do much about it. So, at least from this rule of thumb point of view it is not so easy to come up with a low weight code word for this construction. So, that is the first that we can be happy about in the turbo encoder construction, but that is just the starting point. The main thing will be the decoder.

So, we have constructed this fancy looking encoder. How can we do decoding? So, that is where the idea of concatenation once again helps you. So, if you remember, I pointed out the main idea is, you do not have to decode the overall code. What do you do? You decode the smaller code and then use that information in decoding the other code and it turns out the crucial idea that these days used was this turbo idea. So, you go from 1 decoder to the other decoder and then you do what? You came back to the first decoder. Then you repeat and repeat and repeat and improve your decoding performance. So, once you do that, this code also has capacity approaching performance and bit error rate and all that. So, we will talk little bit more about the encoder and then more details about the decoder tomorrow.

Thank you.