

Incremental Redundancy Via Check Splitting

Moshe Good and Frank R. Kschischang

Dept. of Electrical and Computer Engineering

University of Toronto

{good, frank}@comm.utoronto.ca

Abstract

A new method of creating rateless codes for noisy channels is presented. Unlike puncturing, where every punctured variable disables several checks, or extending, which creates many cycles in the code graph or creates parity checks with insufficient weight, the proposed method uses check splitting to lower the rate of the code. Check splitting operates by replacing a row r of maximum weight in the parity-check matrix with two new rows s_1 and s_2 , of approximately equal weight and with $s_1 \oplus s_2 = r$. This causes the check-node degree-distribution to remain fairly concentrated and prevents cycles from forming as the rate decreases. On Gaussian and Rayleigh fading channels, this scheme performs closer to capacity than Raptor codes, it can use a linear-time-encodable code such as a repeat-accumulate code, and it has nearly constant decoding complexity per information bit per iteration, independent of the effective code rate.

Keywords: automatic repeat request, low-density parity-check codes, punctured codes.

Submitted to *IEEE Transactions on Communications*, March 18, 2007. This paper was presented in part at the 23rd Biennial Symposium on Communications, Kingston, Ontario, May 29–June 1, 2006.

1 Introduction

Error control coding is a mature field. For many channels, once the parameters of a noisy channel have been specified, codes such as turbo codes [1] and low-density parity-check (LDPC) codes [2] and their variants can be designed to be decoded at rates approaching the channel capacity with practical iterative (graph-based) message-passing algorithms. In practice, however, channel parameters may be unknown ahead of time. For example, the transmitter may not know the signal-to-noise ratio (SNR) provided by the channel, particularly when the channel changes with time. Rapidly changing channel conditions occur in many practical communication systems, particularly in wireless systems where, due to multipath fading, the SNR of the channel can change very rapidly as a transceiver is moved over even short distances.

Efficient and reliable communication nevertheless is possible over such time-varying channels, provided that we may implement a feedback channel, over which the receiver can let the transmitter know if and when it has successfully decoded the transmitted message. For example, traditional automatic repeat request (ARQ) schemes operate by appending a cyclic redundancy check (CRC) to the message. The receiver can then know with high probability if the message it has decoded is error free, and request a retransmission if not. Unfortunately, unless the channel is highly likely to be error-free, the throughput (given by P_C , the probability of correct transmission) achieved by traditional ARQ schemes is low.

A better method is the hybrid-ARQ (HARQ) scheme. In the HARQ type I scheme, the data and CRC are encoded with some forward error correction (FEC) code. The throughput of such a scheme is $R \cdot P_C$, where P_C is the probability that we can decode correctly. In theory we should be able to decode correctly whenever $I \geq R$, where I denotes the mutual information between the input and the output of the channel, averaged over the block length of the code; however, in reality we require some extra redundancy ϵ , and so $P_C = \Pr[I(X; Y) \geq R + \epsilon]$.

In case we cannot decode the channel is said to be in outage; the outage probability is $P_O = 1 - P_C$.

The problem with this scheme is when we have a channel with either unknown or highly variable SNR. On the one hand, we would like to pick R to be large to improve throughput on the packets that get decoded. On the other hand, we would like to keep R small enough to ensure a high probability of being able to decode. Regardless of the rate chosen, there will always be some probability (the outage probability P_O) that decoding does not succeed. Even when we can decode, we usually won't be close to capacity.

The obvious conclusion is that given a channel in which the transmitter does not have the channel state information (CSI) we cannot hope to approach capacity using a single-rate code. HARQ type II schemes, also called incremental redundancy schemes, use a rateless code to stay near capacity over a range of rates. These work by first transmitting the systematic (message) bits with only a small number of parity bits. Should decoding fail, additional parity bits are sent and decoding is again attempted. This process continues until successful decoding or some minimum tolerable rate is achieved. Various implementations of this type of scheme have different points at which decoding is attempted. Typically, one has either uniform steps in rate or in the number of symbols sent between decoding attempts.

There has been much recent work in the field of rateless codes [3]. There are two basic approaches, fountain coding and puncturing/extending a mother code. Fountain coding methods are often based on Raptor codes [4], which are themselves based on Luby Transform codes [5]. Most of the recent work, however, seems to focus on puncturing and extending. In [6] puncturing of LDPC and Turbo codes was attempted, with results within 1.5dB from capacity for both the AWGN channel and the fully interleaved Rayleigh fading channel. In [7] it was shown that puncturing from low rates to high rates performs poorly due to the propagation of zero LLR erasure messages in the decoder. Extending was also found to be suboptimal, as the new checks tend to have much lower weight than the old checks. A rate-

compatible code was constructed based on a mix of puncturing and extending to increase the effective SNR range. These ideas continued in [8], where several features were added, such as using a progressive edge growth (PEG) algorithm and puncturing the degree-two variables first. More recently, it was shown in [9] that good puncturing schemes exist, at least as far as density evolution is concerned. This work then continued in [10] where an algorithm was introduced for good puncturing of finite length codes.

1.1 Overview

This work introduces a new method of creating rateless codes. These codes perform well at finite block lengths, and have low complexity. In Section 2 we give the necessary rateless coding background required to understand this work. Section 3 then introduces our new¹ method—called check splitting—of creating rateless codes. In Section 4 we describe the system setup and parameters used in our simulations. Section 4.4 shows how our results compare with other approaches. In Section 5 we discuss implementing our code on nonbinary constellations, though we limit ourselves to one-dimensional pulse amplitude modulation. Finally, in Section 6 we make our concluding remarks.

2 Rateless Codes

The difficulty with HARQ type II systems is the design of its rate-compatible code, a code that can be used over a range of rates. With rate-compatible codes, each higher rate code must be a prefix of each lower rate code, i.e., the lower rate code is the higher rate code with some additional parity bits. While conceptually this doesn't seem to be a difficult task—just

¹We have recently discovered that the idea of check splitting also appears in the independent work of Pisro-Nik and Fekri [11], where it is used to show that puncturing does not result in threshold degradation. The practical results of the present paper are complementary to the theoretical results of [11].

send some more random parities—the additional constraints of operating near capacity with reasonable decoding complexity makes this much more difficult. If we create parity bits of high degree by XORing a large number of message bits together then we get many cycles in the decoding graph as we go to low rates. If we create parity bits of low degree by XORing a small number of message bits together then we may need to send many parity bits to decode, whereas a small number of high-degree parities suffice for high SNR channels. This is the problem we deal with in this work—how to design a good rate-compatible code that works over a large range of rates, from very high rates (> 0.90) down to very low rates (< 0.10).

There are two approaches to creating rate-compatible codes. In the first, usually referred to as a fountain coding, we continuously generate and transmit parity bits by XORing random message bits until we successfully decode. To date, the most popular rateless codes are Raptor codes [4], an implementation of fountain codes. Raptor codes are a concatenation of a high-rate outer code with a Luby Transform (LT) [5] style inner code. LT codes generate a continuous stream of bits, with each transmitted bit generated independently of the other transmitted bits. Each bit is the XOR of d random information bits, where d is a random variable with a robust soliton distribution.

In the second method, we start with some mother code and then puncture/extend it to increase/decrease the rate of the resulting code. However, using this approach we are limited to relatively small changes in the check degree distribution. Suppose we start with a high rate code, with high degree checks, and then attempt to puncture it down to low rates. We are then faced with a question, what check degree should we use for our new checks? If we set the new checks to have low degree, they have insufficient weight to affect the decoder behavior, as all the other checks have many more edges emanating from them. We would need many such checks to allow us to use an even slightly inferior channel than the one for which the mother code was designed. It thus seems that we are forced to create high degree checks. While this works well at high code rates, as the code rate drops our decoding

graph quickly becomes highly connected with many small cycles. These cycles then allow many incorrect assumptions to loop around and reinforce themselves.

Suppose we start with a low rate code and try to puncture it to high rates. This causes many checks, the ones with punctured variables, to behave poorly. Checks with a single punctured variable start off by producing just a single message, the one to the punctured variable. Checks with several punctured variables start off unable to do *anything*. There is a $\frac{P \cdot \binom{N-P}{d_c-1}}{\binom{N}{d_c}}$ probability that a check has a single puncture, and a $1 - \frac{P \cdot \binom{N-P}{d_c-1} + \binom{N-P}{d_c}}{\binom{N}{d_c}}$ probability that a check has several punctures, where N is the size of the unpunctured codeword, d_c is the check node degree, and P is the number of punctured positions. These formulas show that a large fraction of checks become useless as the rate of the punctured code increases.

Puncturing is essentially the approach we follow in this paper. However, unlike traditional puncturing schemes, our mother code is structured in a manner so that only degree-two variable nodes are punctured; furthermore, when such a variable node is punctured, the two affected checks are combined into a single check of higher degree. This ensures that all our checks are operating at full effectiveness.

3 Check Splitting

3.1 Motivation

It is known that high rate codes require high degree checks to do well. Low degree checks do not have enough edges to make the decoding graph well connected at high rates. Similarly, it is known that low rate codes require low degree checks to do well with belief-propagation decoders. High degree checks at low rates cause the decoding graph to be overly connected, causing many unnecessary cycles. Indeed, by querying R. Urbanke's `ldpcopt` web site [12] for threshold optimized irregular LDPC codes over a spectrum of rates, it is easy to see that

the average right degree is low for low rate codes and increases monotonically as the rate of the code increases.

These two facts lead us to ask: is it possible to have a code that has high degree checks at high rate yet has low degree checks at low rate? At first glance, this seems impossible. A rateless code must have the property that every high rate code is a prefix of every low rate code. It follows that while we can add new constraints as the code rate drops, all these additional constraints must involve the newly sent information bits/bytes. The older constraints, however, must remain valid regardless of the effective code rate. This *seems* to imply that the old check degrees are immutable.

3.2 Check Splitting and their Properties

The trick is to use equivalent codes. It is clear that a parity check matrix H has the same null space (parity constraints) as $H' = AH$, where A is any square nonsingular matrix. We can therefore change older check degrees by combining them with new checks. Since we wish to decrease the average check degree while keeping a concentrated check degree distribution, we should ensure that the sum of an old check with a new check has about the same degree as the the new check. This can be accomplished by simply setting the new check to contain a subset of the old check whose degree equals half the weight of the old check.

We can describe this process, which we call *check splitting*, (see also [11]) more rigorously.

First, let us define H to be the current parity-check matrix of size $m \times n$, and let $V(i) = \{j : h_{ij} = 1\}$ be the set of columns of H containing ones in row i .

In our approach, each time we wish to extend H by one bit, we do the following:

1. Append a column of zeros to H to make room for another variable.
2. Select a row i from H such that $|V(i)| \geq |V(j)|$, $\forall j \in 1 \dots m$. This row represents a

check of maximum degree.

3. Partition $V(i)$ into $V' \subseteq V(i)$ and $V'' \subset V(i)$ such that $V' \cup V'' = V(i)$, $V' \cap V'' = \emptyset$, and $|V'| - |V''| \in \{0, 1\}$.
4. Append a row to H so that $V(m + 1) = V'' \cup \{n + 1\}$.
5. Change row i so that $V(i) = V' \cup \{n + 1\}$. This removes all the cycles that adding row $m + 1$ created.

We call this process *check splitting*, as it converts a check of degree d into two checks, one of degree $\lfloor d/2 + 1 \rfloor$ and one of degree $\lceil d/2 + 1 \rceil$.

For example, suppose we wish to check split a single check of degree eight into two checks of degree five. Ignoring the columns containing zeros in both the old and new checks, we would have

$$\begin{aligned}
 H_1 &= \left[\begin{array}{c} 11111111 \end{array} \right] \\
 H_2 &= \left[\begin{array}{c|c} 11111111 & 0 \\ \hline 00001111 & 1 \end{array} \right] \\
 H_{2,EQ} &= \left[\begin{array}{c|c} 11110000 & 1 \\ \hline 00001111 & 1 \end{array} \right],
 \end{aligned}$$

where $H_{2,EQ}$ is an equivalent parity check matrix to H_2 . A graphical representation of this process can be seen in Fig. 1.

Check splitting causes *no* new cycles in the decoding graph. This is in great contrast to classical forms of extending, which tend to produce many short cycles, especially when based on high rate mother codes. Another interesting property is that check splitting does

not increase decoding complexity. When using the sum-product decoding algorithm, each check node of degree d_c requires $3 \cdot (d_c - 2)$ check node operations, often implemented as table look-ups. Each variable node of degree d_v requires $2 \cdot (d_v - 1)$ additions. When we split a check node of degree d_c , we end up with two check nodes of degree $\lfloor d_c/2 + 1 \rfloor$ and $\lceil d_c/2 + 1 \rceil$. These require a total of $3 \cdot (\lfloor d_c/2 + 1 \rfloor - 2) + 3 \cdot (\lceil d_c/2 + 1 \rceil - 2) = 3 \cdot (d_c - 2)$ operations, which is exactly what we had before. As for the variable nodes, all the newly created variable nodes are of degree two, while all the old variable nodes keep their old degrees. Since the computation required per variable node is much smaller than what is needed per check node, the increasing number of degree two variable nodes doesn't have a noticeable effect on decoding complexity.

3.3 Problems with Check Splitting and Solutions

One issue that arises when using check splitting is that of the stability condition [13]. The stability condition for the binary-input AWGN channel states that if $\lambda_2 > \frac{e^{1/2\sigma_n^2}}{\rho'(1)}$ then the probability of bit error cannot go to zero under a “parallel” message passing schedule (where variables send message to checks in parallel, and then checks send messages to variables in parallel). However, if $\lambda_2 < \frac{e^{1/2\sigma_n^2}}{\rho'(1)}$ then there exists a channel with low enough noise over which the probability of bit error goes to zero as we increase the number of cycle free decoding iterations. Since check splitting always introduces new variables of degree two, we will always run into problems with the stability condition as the code rate drops.

To further compound this problem, it is usually impractical to change the decoding graph as the effective rate drops. To solve this problem we can use the lowest rate graph all along, and view unreceived bits as being punctured. But, this means that we will violate the stability condition from the very beginning. Moreover, having a large number of punctured bits in the decoding graph will seriously impact decoding performance as explained in [7].

We get around the problem of the stability condition by altering the message-passing schedule, so that the analysis of [13] is irrelevant. As mentioned, all the check-splitting variables are of degree two. This creates chains of checks connected by those degree two variables. We can then use the forward-backward algorithm on these chains to fully propagate the information over them. The full message passing schedule would then be the following.

- Pass messages from all variable nodes to the check nodes
- Perform the forward-backward algorithm along every chain of checks connected by the check-splitting degree-two variables. Every check is split, so there are no checks not in a chain.
- Repeat until we are have decoded successfully.

The forward-backward algorithm has a check-combining effect, whereby the chain of checks acts as a single higher degree check. This is due to the forward-backward algorithm producing exact log-likelihood ratios, even when we have many punctures in the chain, as there are no cycles involved. For example, if we start with a degree ten check and split it all up, we end up with eight checks of degree three connected in a chain. If we then puncture the seven connecting variables and perform the forward-backward algorithm, we will produce the same messages to the original non-punctured variables as would have been produced by the original degree ten check. This solves both the problem of having a large number of variables punctured as well as the problem with the stability condition.

4 System Setup and Results

4.1 Choice of Mother Code

In our setup we decided to use check splitting on an irregular repeat-accumulate code [14]. We chose a repeat-accumulate code instead of a standard LDPC code for several reasons. First of all, a repeat-accumulate code has a very simple, linear time complexity, systematic encoder. Being systematic, it is also easy to get our decoded information out of the decoder. Secondly, it is easy to puncture the accumulator, without loss of performance, all the way up to a code rate of about one. This could not be done with a standard LDPC mother code, as explained in [7]. The most important reason, however, is that the accumulator structure makes check splitting very simple to implement.

As explained in the previous section, every time we split a check we produce a degree two variable. Furthermore, we need to use the forward-backward algorithm over these variables. A repeat-accumulate code has many degree two variable nodes in its accumulator, and we generally use the forward-backward algorithm over them. This allows us to use a simple repeat-accumulate code as the low rate decoding graph. We can then puncture the accumulator to get a higher rate code, just as we would puncture the degree-two check-splitting variables when using an LDPC mother code.

4.2 Puncturing Methods

The question then is, how do we puncture the accumulator to maximize performance? If we start with a repeat-accumulate code of right-degree one and then puncture every other bit in the accumulator, we end up with a repeat-accumulate code of right-degree two, as illustrated in Fig. 2. Again if we were to puncture every other remaining bit, we would have a repeat-accumulate code code of right-degree four. Should we continue this puncturing

process, we would end up with a code of rate $K/(K + 1) \approx 1$, containing just a single check. Together with the systematic bits, this parity check constitutes the initial transmission.

We can lower the effective rate by unpuncturing the accumulator (which, for a repeat-accumulate code, is effectively check-splitting) until we are able to decode. If all our checks have been split down to right-degree one checks and we still are unable to decode, we use repetitions to increase the effective SNR until we are able to decode. See Table 1 for an example of the order in which we send the accumulator variables. Basically, the algorithm can be summarized as follows.

```

Let  $C = \lceil \log_2 N \rceil$ .
for ( $s = 2^C; s \geq 1; s = s/2$ ) do
  for ( $i = s; i \leq N; i = i + 2s$ ) do
    Send variable  $i$ .
  end for
end for

```

It should be noted that this unpuncturing order keeps checks next to other checks of the same degree. This allows us to have two subsections of accumulator, one half with low degree checks and one half with high degree checks. This is a good thing, as the subsection with low degree checks will produce highly reliable messages. Should we interleave the checks of differing degrees, such as by sending the parity bits in reversed bit-reversed order, we would not get highly reliable messages from any part of the accumulator. This is because the high degree checks would be passing unreliable information to the low degree checks.

4.3 Building a good code

We designed a degree distribution via extrinsic information transfer charts with the Gaussian approximation for repeat-accumulate codes, as described in [15]. We decided to optimize the

Variable #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Send Posn.	9	5	10	3	11	6	12	2	13	7	14	4	15	8	16	1
Rx. Order	16	8	4	12	2	6	10	14	1	3	5	7	9	11	13	15

Table 1: The order in which we transmit variables. We bisect chunks of punctured variables, effectively check splitting on the RA code.

code for a binary input additive white Gaussian noise (BIAWGN) channel with a Rayleigh fading parameter, such that the average SNR was one. This corresponds to an ergodic channel capacity of 0.4 bits/symbol per channel use. Additionally, we constrained our system to have a maximum of four variable degrees with the highest variable degree to be no more than ten. This gave us the left degree distribution $\lambda(x) = 0.060x + 0.241x^2 + 0.266x^7 + 0.433x^9$. The right degree will obviously change with the effective code rate. We also tried optimizing the mother code, with no consideration given to its punctured performance. This gave us $\lambda(x) = 0.0365x + 0.2712x^2 + 0.023x^8 + 0.670x^9$. As expected, the first code performs slightly better at higher rates, while the second does better at rates near that of the mother code.

We built our parity check matrix using a variant of the ACE PEG algorithm [16]. As a first constraint on the code graph, we obviously do not allow cycles of length two (double edges). Furthermore, we do not allow cycles of length six or less containing only variables of degree three and/or two, which are the only variables in which we found decoding errors during simulations. We also don't allow cycles of length eight or less containing only degree two variables, as cycles with only degree two variables have no extrinsic edges. These graph sub-cycles were selected due to their large negative impact on performance and ease of removal. Cycles with high degree variable nodes don't seem to significantly impact performance, as they have high connectivity to extrinsic information, and anyway are often impossible to remove when dealing with short block lengths.

4.4 Results

The two channels on which we focus are the AWGN channel with unknown SNR, and the fully interleaved Rayleigh flat fading channel, in which we assume the fading parameter is independent and identically distributed over each symbol. Our code is constructed as follows. First, we append a 32-bit cyclic redundancy check (CRC) to our data. This will let us know when we have decoded successfully. Then, we encode the data and CRC with a high rate systematic linear-time encodable IRA code. We first attempt to decode when the code rate is equal to the mutual information of the channel. We then continue trying to decode every 0.1 dB farther away from the Shannon limit.

We look at three check-node degree-distributions, and compare their performance in Fig. 3. First, we look at performance of a concentrated degree distribution. While this is impossible to achieve in practice, it gives a simple upper bound on performance for any practical distribution. Secondly, we look at a distribution with all check nodes concentrated around degrees 2^k and 2^{k+1} , where k is a positive integer. This corresponds to the simplest possible method of splitting the checks, but has high variance in the check node degree distribution at some rates. Lastly, we look at the situation in which we have three degree concentrations, half the nodes concentrated in 2^k or 2^{k+1} and half in $3 \cdot 2^k$ or $3 \cdot 2^{k+1}$, where again k is a positive integer. Because we always split a check of maximum degree we end up with a maximum of three check node degree concentrations at any rate. As expected, given a specific SNR, the lower the variance of the check degrees, the higher the rate.

We then ran simulations to see how our code would perform in practice. Simulation results are shown in Figs. 4 and 5. Interestingly enough, changing several of the code parameters doesn't seem to significantly impact the performance, as can be seen in Fig. 4.

On an BIAWGN channel of capacity $1/2$ ($\text{SNR} = E_s/\sigma^2 = 0.185$ dB), it was found in [17] that using a Raptor code allowed 9500 bits of information to be sent, with only 20737

bits transmitted on average. We compared this to our code with two-point check-degree concentrations and $\lambda(x) = 0.060x + 0.241x^2 + 0.266x^7 + 0.433x^9$. Starting with a rate 0.857 mother code based on IRA codes, we find that we can decode with only 20380 bits on average, less than 0.5 dB from the SNR needed to achieve such a rate at capacity.

We then compared our results to Raptor codes, as they seem to be the best available rateless codes when dealing with a large dynamic SNR range. As can be seen in Figs. 6 and 7, our codes outperform Raptor codes for SNR values above about -6 dB. However, Raptor codes do better at lower rates. We believe this is due in large part to our code degenerating into repetition coding below the rate of the mother code. For our code, this happens at a rate of about 0.15, which occurs at roughly -5.5 dB. We should be able to improve performance at these low rates by traditional extending, which is easy to do as our check nodes would all be split down to degree three.

5 Higher Order Constellations

5.1 Background

Using binary/quaternary phase shift keying (BPSK/QPSK) for our modulation scheme limits us to one bit per channel use per dimension. For high SNR channels, we can do better by using higher order modulation schemes. One method used for higher order constellations is multilevel coding (MLC) [18]. In MLC, each bit level contains a code that is decoded independently of the code on any other layer. This can be used in conjunction with multistage decoding [18](MSD) to allow us to approach capacity [19]. In MSD, after each layer is decoded we use the decoded information to better estimate the LLRs of the bits on the undecoded layers.

We can also do parallel independent decoding (PID) of each component code, where we do

not use any information from any previously decoded code to help the other decoders. This lets us perform the decoding for each layer in parallel with the decoding of the other layers. The tradeoff is that we are bounded away from capacity. Using PID requires a good choice of labelling to ensure we stay close to capacity. The best is Gray labelling, with which we can achieve near maximum mutual information [19], at least when the code rate is 1/2 or higher. See Fig. 8 to compare the capacities of MSD and PID with Gray labelling for a 4-PAM system.

Another method of using higher order constellations is based on bit-interleaved coded modulation (BICM) [20]. In this method we use a single long codeword spread over the several bit levels. This allows us to have longer codewords, which improves performance of LDPC codes. Similar to MLC, this code can be decoded in two ways. In the first, we assume LLRs coming from the channel are fixed. These channel LLRs are similar to those seen when using a PID MLC approach, and as such require Gray labelling to perform well. This was the approach used in [20] and in our work.

We can also pass messages through symbol nodes while decoding, to get conditional channel LLRs to propagate through the decoder. The symbol node is a modified check node corresponding to the modulation used on the transmitted symbol, which generates new channel LLRs based on our current beliefs of the values of the bits making up a symbol. For example, suppose we have a 4-PAM system with symbol mapping

(b_1, b_2)	01	00	10	11
Tx. Value	-3	-1	+1	+3

then the LLR from our symbol node to b_1 would be

$$u_1 = \log \left(\frac{\exp\left(\frac{-(x+1)^2}{2\sigma^2}\right) \cdot P(b_2 = 0) + \exp\left(\frac{-(x+3)^2}{2\sigma^2}\right) \cdot P(b_2 = 1)}{\exp\left(\frac{-(x-1)^2}{2\sigma^2}\right) \cdot P(b_2 = 0) + \exp\left(\frac{-(x-3)^2}{2\sigma^2}\right) \cdot P(b_2 = 1)} \right),$$

where x is the received value on the channel and $P(b_2)$ is calculated based on the LLR from b_2 to the symbol node. We get $P(b_2 = 1) = (1 + \exp(v))^{-1}$ and $P(b_2 = 0) = (1 + \exp(-v))^{-1}$,

where v is the LLR from b_2 . These conditional messages are similar to MSD, and do not require a specific labelling scheme. It is much simpler, however, to not have any symbol nodes, and to assume the channel LLRs are fixed.

5.2 Results

We have based our simulations on the standard BICM approach described. This means that we cannot hope to achieve performance above that of the ideal PID curve. We assume that if we were to pass messages over the symbol nodes, then we would track the MSD performance curve instead of tracking the PID curve. Results for the fixed LLR approach can be seen in Fig. 8 for the case of a 4-PAM signalling constellation. Very similar results (not shown), with a maximum rate of 3 bits per symbol, were obtained for the case of 8-PAM.

One way to increase performance when using fixed channel LLRs is to use a bit-loading algorithm [21]. With a bit-loading algorithm we start our transmission using a large constellation, hoping for a high SNR. Should we not decode within the expected time frame (for a high SNR channel), we realize that the SNR is lower than we had hoped, and so begin using a smaller constellation. Smaller constellations have less SNR gap between their MSD and PID capacity curves, so we can actually increase rate with a smaller constellation, assuming we are utilizing the PID approach and have low SNR. Even when using MSD there are advantages to using a small constellation, such as lower complexity, and improved performance for finite length codes. The improved performance is presumably due to having fewer incorrect messages propagating through the decoder.

6 Conclusion

Check node splitting solves the problems associated with rateless codes that can achieve high effective rates. This allows us to make simple codes that perform very well over a large SNR range with low complexity. Our rateless codes based on irregular repeat accumulate codes with check node splitting outperform Raptor codes for AWGN channels at most rates. With slight modification (allowing extending at low rates) we conjecture that they may be able to outperform Raptor codes at *all* rates. They also have the advantage of being systematic, linear-time encodable, and have $O(N)$ complexity per decoding iteration, independent of the operating rate of the code, and hence the number of bits received.

We now discuss some areas for future work on this topic. It would be interesting to see how a combined check-splitting and extending algorithm would perform. We know that we are getting suboptimal performance in the low SNR range due to repetitions, which can be avoided with extending. We should also be able to improve performance at higher SNR ranges by mixing extending, which would allow us to change the variable node degree distribution, and check-splitting, which would allow us to change the check node degree distribution. We can then follow some ‘path’ of good degree distributions $(\lambda(x), \rho(x))$ along a wide SNR range. This degree distribution path concept is discussed in [22], but the approach taken in that paper is severely limited as the authors do not utilize check-splitting to keep the check degree distribution concentrated. We believe that this combined approach will allow us to create capacity-approaching and capacity-tracking rateless codes.

References

- [1] C. Berrou, A. Glavieux, and P. Thitimajshima, “Near Shannon limit error-correcting coding and decoding: Turbo-codes (1),” *Proc. IEEE Int. Conf. Communications*, pp. 1064–1070, 1993.
- [2] R. Gallager, “Low-density parity-check codes,” *IEEE Transactions on Information Theory*, vol. 8, pp. 21–28, Jan 1962.
- [3] D. Mandelbaum, “An adaptive-feedback coding scheme using incremental redundancy,” *IEEE Transactions on Information Theory*, vol. 20, pp. 388–389, May 1974.
- [4] A. Shokrollahi, “Raptor codes,” *International Symposium on Information Theory*, p. 36, 2003.
- [5] M. Luby, “LT codes,” *The 43rd Annual IEEE Symposium on Foundations of Computer Science*, pp. 271–280, 2002.
- [6] R. Mantha and F. Kschischang, “A capacity-approaching hybrid ARQ scheme using turbo codes,” *Global Telecommunications Conference*, vol. 5, pp. 2341–2345, 1999.
- [7] J. Li and K. R. Narayanan, “Rate-compatible low density parity check codes for capacity-approaching ARQ schemes in packet data communications,” *Int. Conf. on Comm. Internet and Info. Tech*, Nov 2002.
- [8] M. Yazdani and A. H. Banhashemi, “On construction of rate-compatible low-density parity-check codes,” *IEEE International Conference on Communications*, pp. 430–434, June 2004.
- [9] J. Ha, J. Kim, and S. McLaughlin, “Rate-compatible puncturing of low-density parity-check codes,” *IEEE Transactions on Information Theory*, vol. 50, pp. 2824–2836, Nov 2004.

- [10] J. Ha, J. Kim, D. Klinc, and S. McLaughlin, “Rate-compatible punctured low-density parity-check codes with short block lengths,” *IEEE Transactions on Information Theory*, vol. 52, pp. 728–738, Feb 2006.
- [11] H. Pishro-Nik and F. Fekri, “Results on punctured low-density parity-check codes and improved iterative decoding techniques,” *IEEE Transactions on Information Theory*, vol. 53, pp. 599–614, 2007.
- [12] R. Urbanke, “ldpcopt: a degree distribution optimizer for irregular ldpc code ensembles.” online at lthcwww.epfl.ch/research/ldpcopt.
- [13] T. Richardson, A. Shokrollahi, and R. Urbanke, “Design of capacity-approaching irregular low-density parity-check codes,” *IEEE Transactions on Information Theory*, pp. 619–637, Feb 2001.
- [14] H. Jin, A. Khandekar, and R. McEliece, “Irregular repeat-accumulate codes,” *Proc. 2nd International Symposium on Turbo Codes*, pp. 1–8, 2000.
- [15] S. ten Brink and G. Kramer, “Design of repeat-accumulate codes for iterative detection and decoding,” *IEEE Transactions on Signal Processing*, vol. 51, pp. 2764–2772, November 2003.
- [16] T. Tian, C. Jones, J. D. Villasenor, and R. D. Wesel, “Construction of irregular LDPC codes with low error floors,” *IEEE International Conference on Communications*, vol. 5, pp. 3125–3129, May 2003.
- [17] R. Palanki and J. S. Yedidia, “Rateless codes on noisy channels,” *International Symposium on Information Theory*, p. 38, 2004.
- [18] H. Imai and S. Hirakawa, “A new multilevel coding method using error correcting codes,” *IEEE Transactions on Information Theory*, vol. 23, pp. 371–377, May 1977.

- [19] U. Wachsmann, R. F. Fischer, and J. B. Huber, “Multilevel codes: Theoretical concepts and practical design rules,” *IEEE Transactions on Information Theory*, vol. 45, pp. 1361–1391, July 1999.
- [20] G. Caire, G. Taricco, and E. Biglieri, “Bit-interleaved coded modulation,” *IEEE Transactions on Information Theory*, vol. 44, pp. 927–946, May 1998.
- [21] M. Ardakani, T. Esmailian, and F. R. Kschischang, “Near-capacity coding in multicarrier modulation systems,” *IEEE Transactions on Communications*, vol. 52, pp. 1880–1889, November 2004.
- [22] D. Bi and L. Prez, “Rate-compatible low-density parity-check codes with rate-compatible degree profiles,” *Electronics Letters*, vol. 42, pp. 41–43, January 2006.

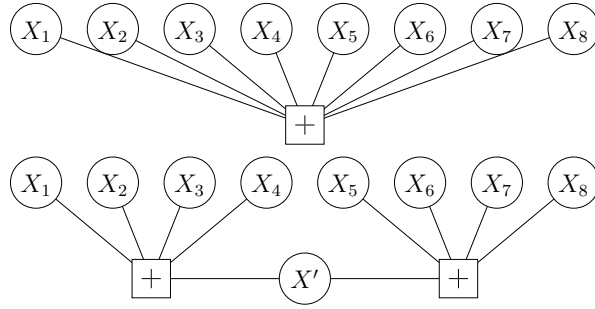


Figure 1: Check node splitting. The check node is replaced with two checks of smaller degree and a connecting variable X' .

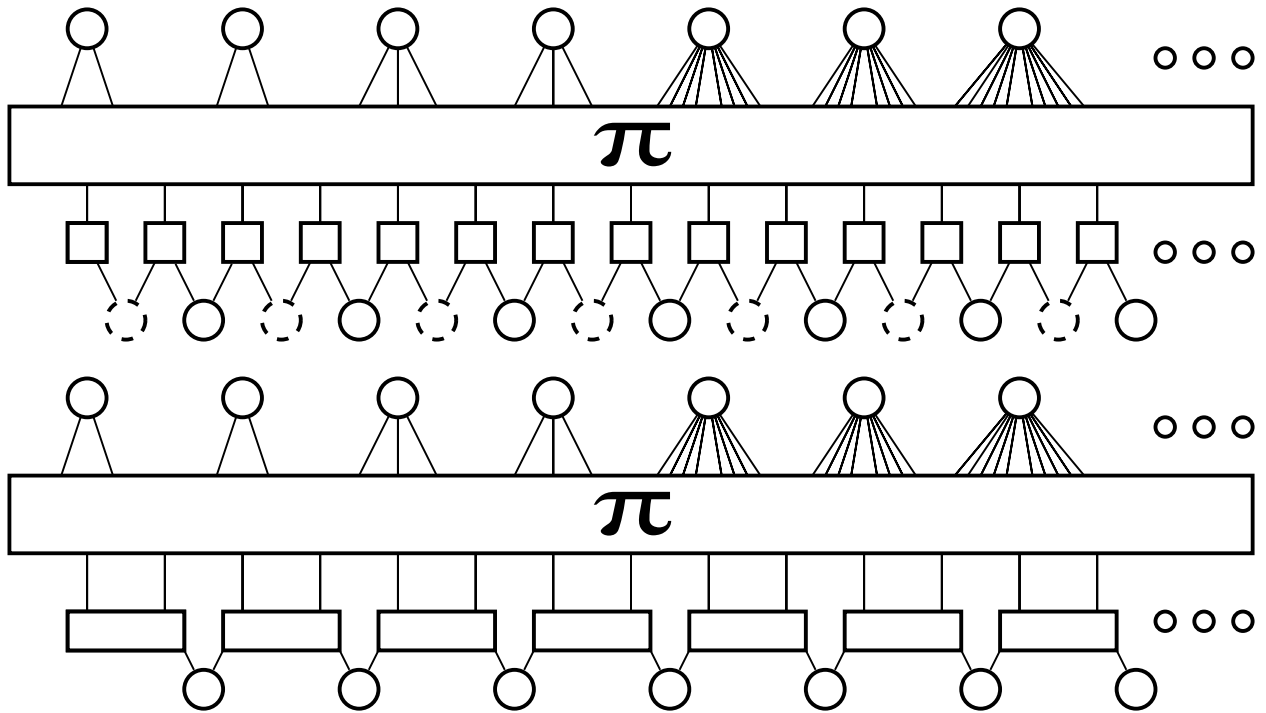


Figure 2: Puncturing a right degree one IRA code and the equivalent right degree two code. The right degree is the number of edges from the accumulator that each check is connected to.

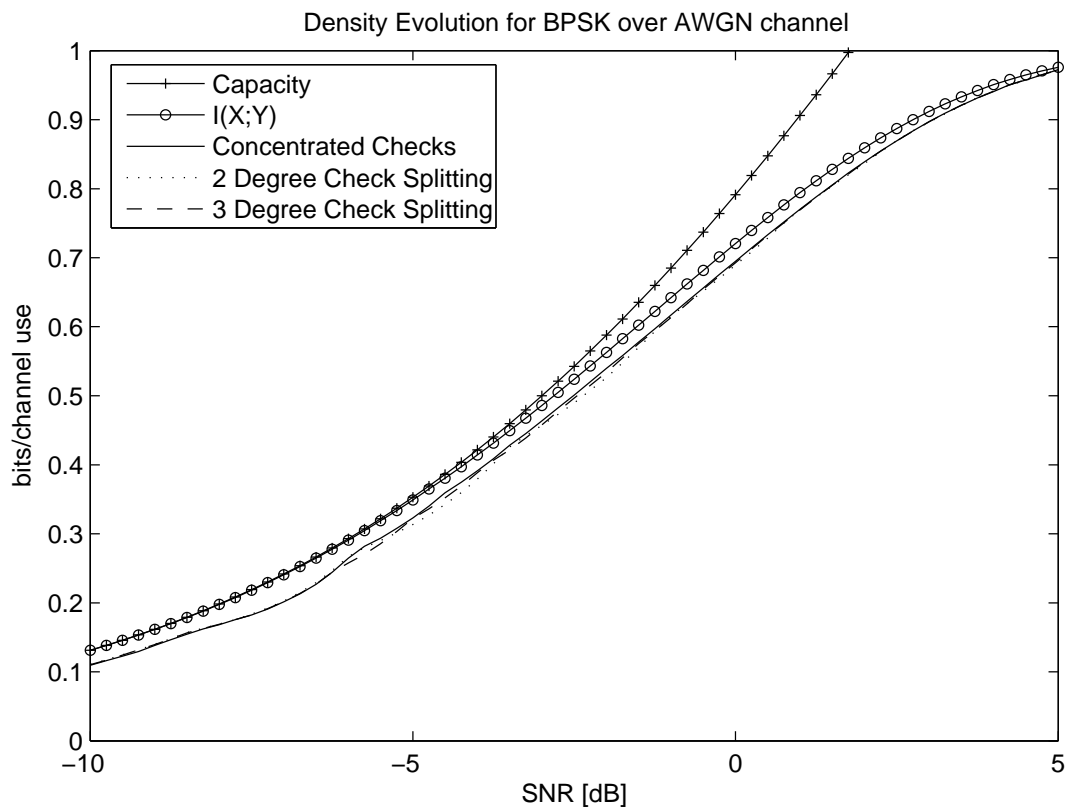


Figure 3: Density evolution results for our code over a large SNR range.

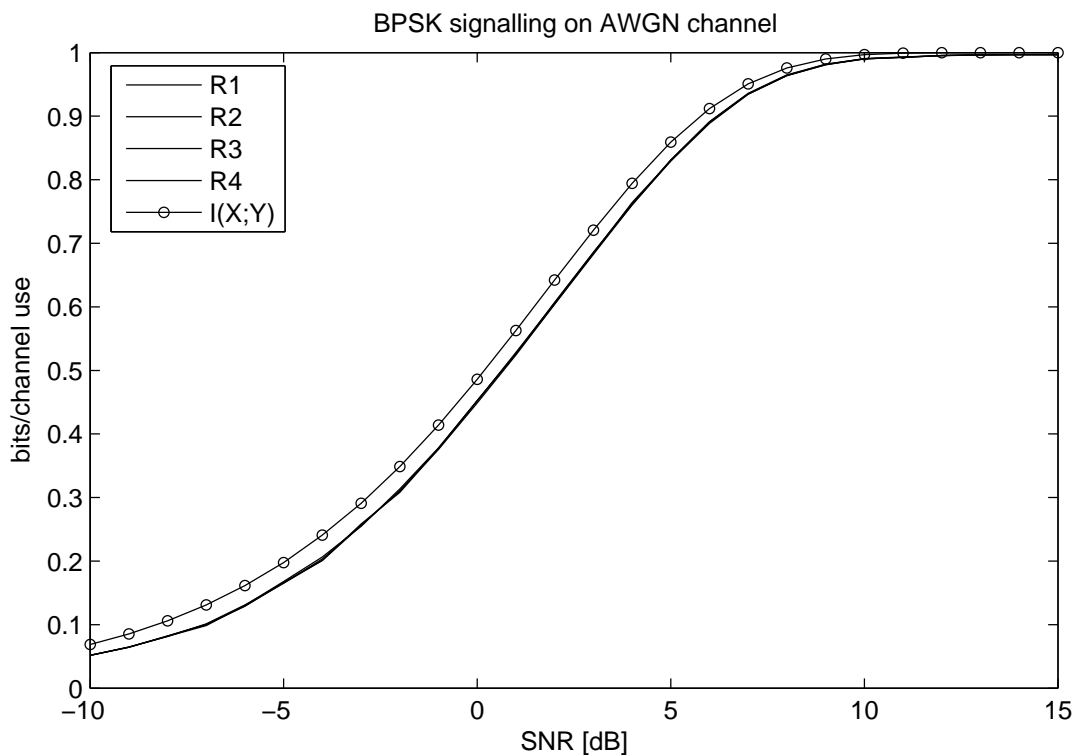


Figure 4: Actual results for various parameter choices. Step size is 0.005 bits/symbol with a maximum of 100 iterations per step. All simulations have $\lambda(x) = 0.060x + 0.241x^2 + 0.266x^7 + 0.433x^9$ except R2, which has $\lambda(x) = 0.35x^2 + 0.08x^7 + 0.57x^9$. All simulations have a mother code of rate 0.923 except R3, which has rate 0.75. All simulations use a two-point check-degree concentration except R4, which uses a three-point check-degree concentration. None of these changes seem to affect the actual performance.

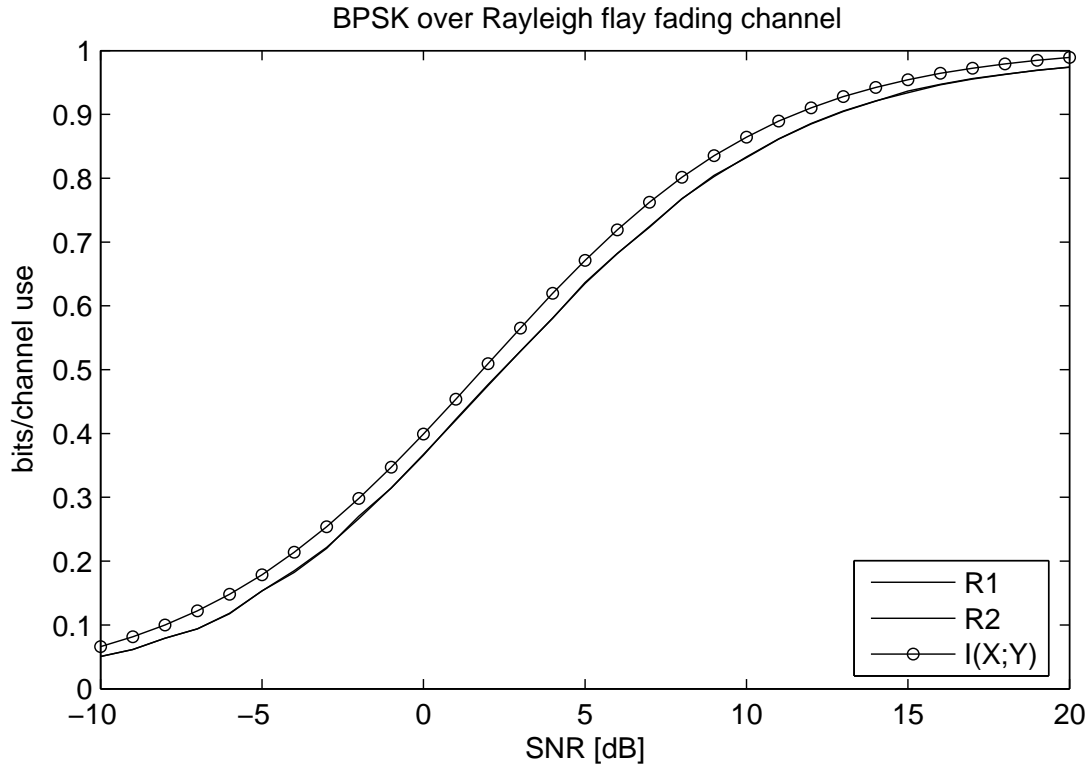


Figure 5: Actual results for various parameter choices. Step size is 0.005 bits/symbol with a maximum of 100 iterations per step. Both simulations have $\lambda(x) = 0.060x + 0.241x^2 + 0.266x^7 + 0.433x^9$ and a mother code of rate 0.923. R1 uses a two-point check-degree concentration, while R2 uses a three-point check-degree concentration.

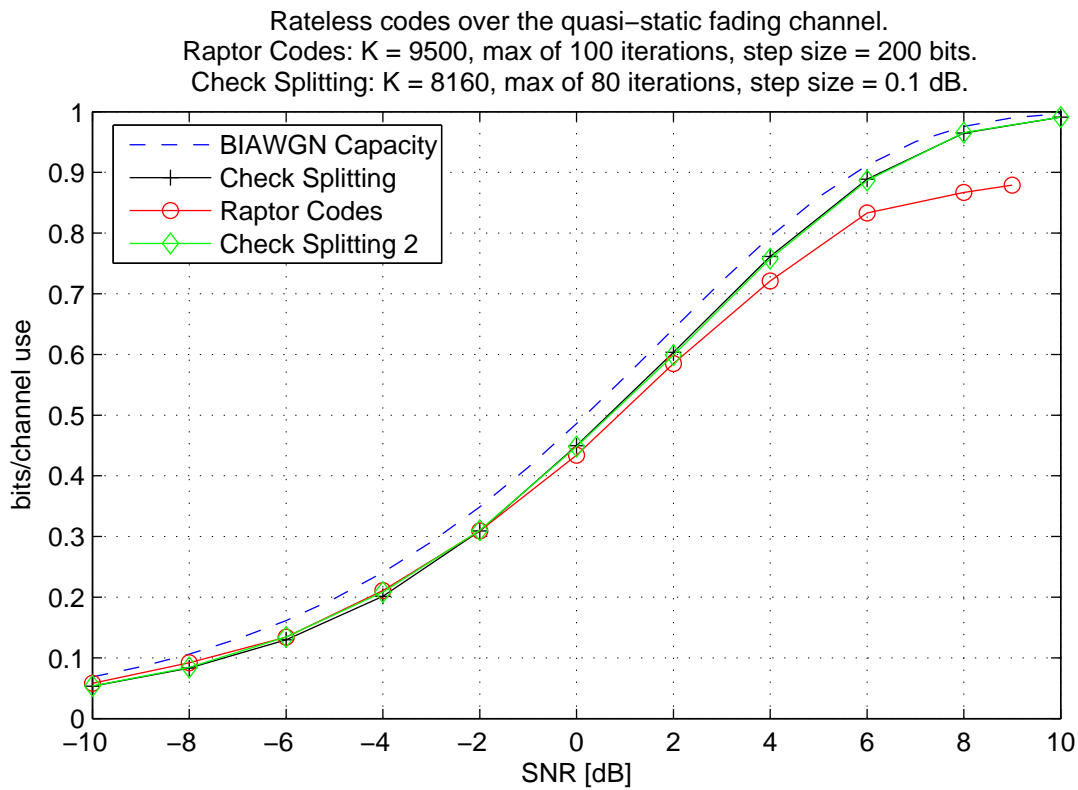


Figure 6: The ‘Check Splitting’ curve has $\lambda(x) = 0.060x + 0.241x^2 + 0.266x^7 + 0.433x^9$. The ‘Check Splitting 2’ curve has $\lambda(x) = 0.32x^2 + 0.68x^9$. The ‘Raptor Codes’ curve has an LT distribution of $\Omega(x) = 0.05x + 0.5x^2 + 0.05x^3 + 0.25x^4 + 0.05x^6 + 0.1x^8$ and a regular (3,30) LDPC precode.

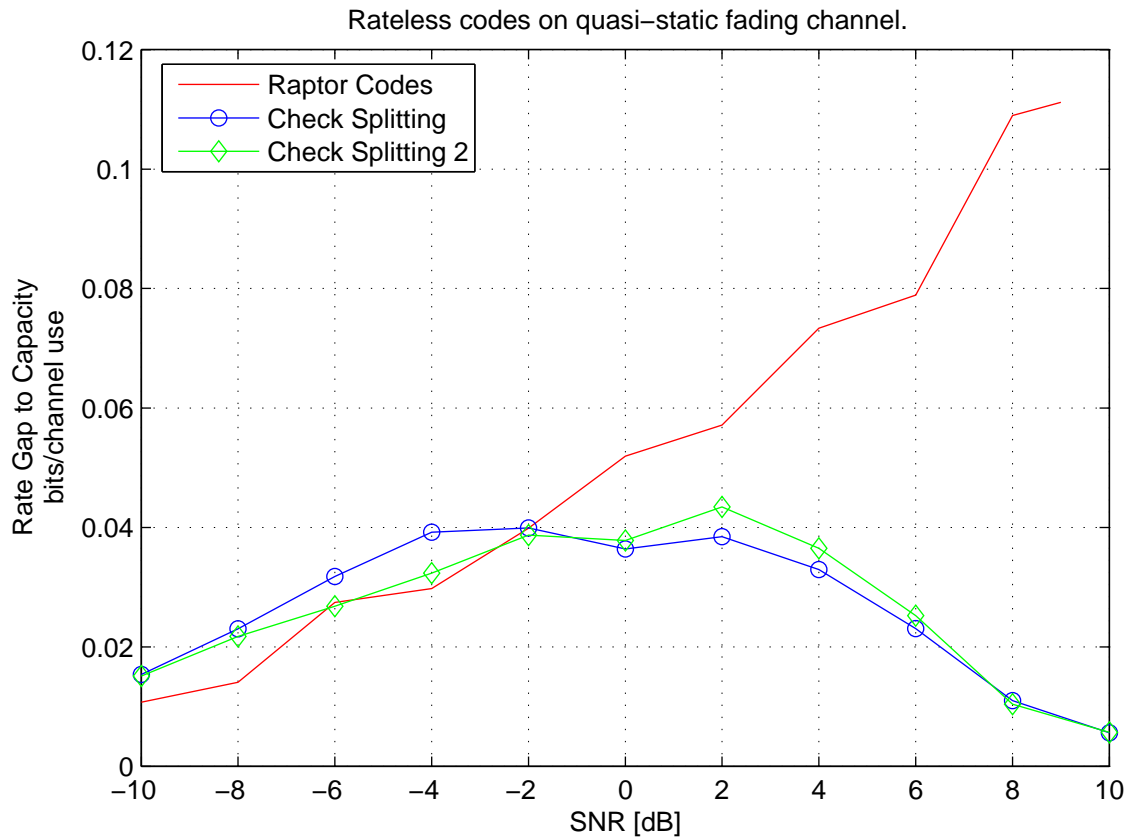


Figure 7: The ‘Check Splitting’ curve has $\lambda(x) = 0.060x + 0.241x^2 + 0.266x^7 + 0.433x^9$. The ‘Check Splitting 2’ curve has $\lambda(x) = 0.32x^2 + 0.68x^9$. The ‘Raptor Codes’ curve has an LT distribution of $\Omega(x) = 0.05x + 0.5x^2 + 0.05x^3 + 0.25x^4 + 0.05x^6 + 0.1x^8$ and a regular (3,30) LDPC outer code.

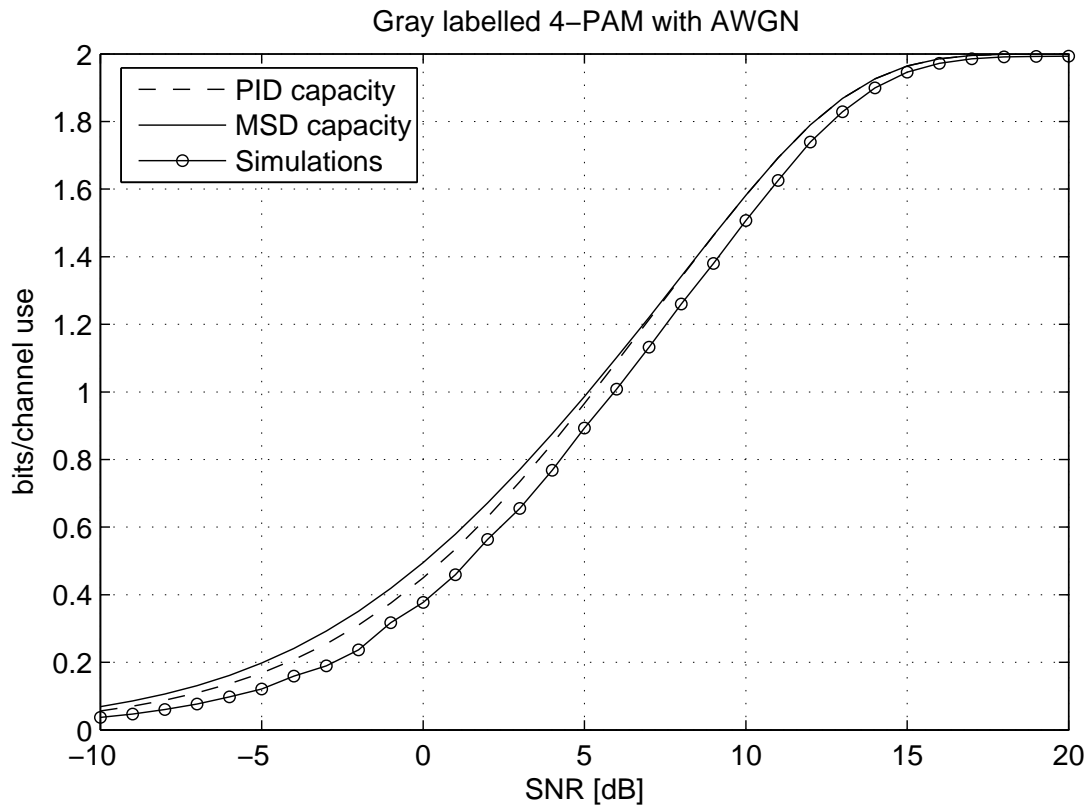


Figure 8: Comparison between MSD and PID capacities for MLC with 4-PAM signalling and Gray labelling. Simulations use static channel LLRs (PID approach). Step size is 0.1 dB with a maximum of 100 iterations per step.

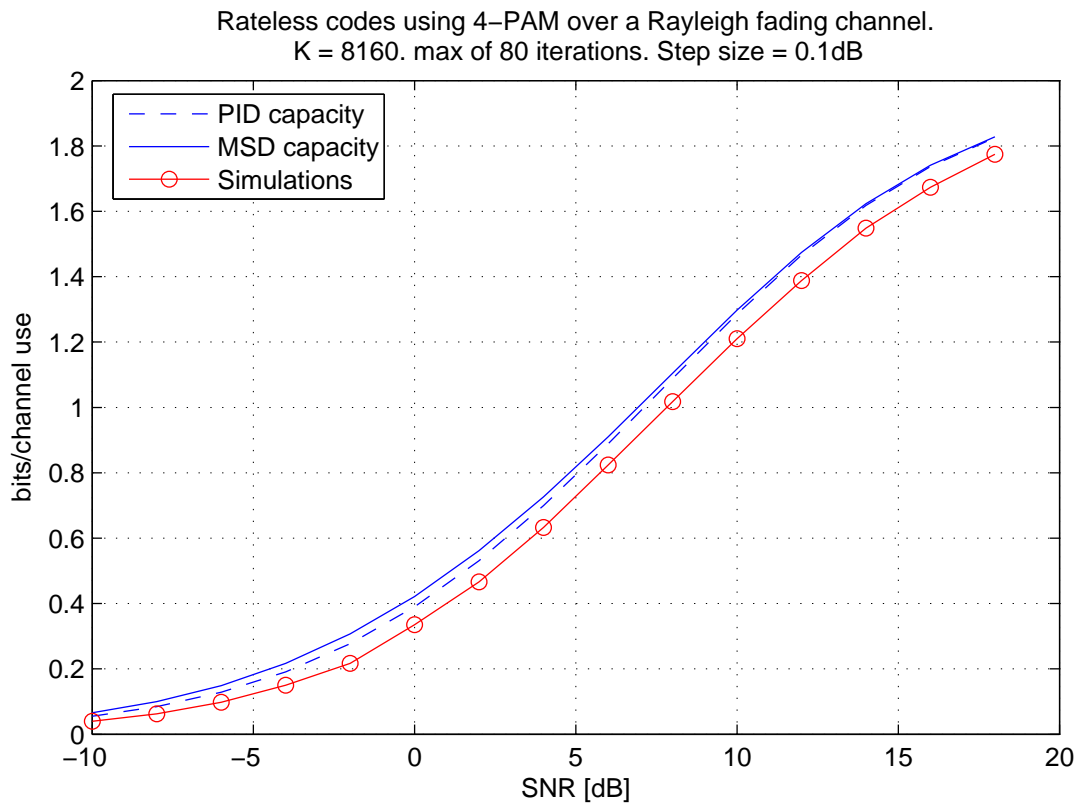


Figure 9: Comparison between MSD and PID capacities for MLC with 4-PAM signalling and Gray labelling. Simulations use static channel LLRs (PID approach). Step size is 0.1 dB with a maximum of 80 iterations per step.