TELKOMNIKA, Vol.10, No.4, December 2012, pp. 759~770
ISSN: 1693-6930
accredited by DGHE (DIKTI), Decree No: 51/Dikti/Kep/2010

Effects of Puncturing Patterns on Punctured Convolutional Codes

Lydia Sari
Electrical Engineering Department, Faculty of Engineering, Atma Jaya Catholic University
Jln. Jend.Sudirman Kav. 51, Jakarta 12930, Indonesia Ph. 62-21-5708826
e-mail: lydia.sari@atmajaya.ac.id

Abstrak

Kode konvolusional punctured telah digunakan secara luas dalam sistem telekomunikasi karena hemat bandwidth dan lebih sederhana dibandingkan kode non-punctured, namun tetap memiliki kinerja yang baik. Analisa kinerja kode konvolusional punctured dapat disederhanakan menggunakan kode ekivalen non-punctured. Dalam makalah ini diajukan kode konvolusional punctured baru dengan rate 3/8, 3/7 dan 3/6, dan kinerjanya dianalisa dengan terlebih dahulu mengkonstruksi kode ekivalen non-punctured. Hasil simulasi menunjukkan bahwa pola puncturing yang berbeda untuk laju kode yang sama akan mempengaruhi kinerja kode. Penelitian lebih lanjut menunjukkan bahwa puncturing pada bit-bit yang bersebelahan perlu dihindari karena berpotensi menurunkan kinerja kode, seperti diindikasikan dengan penurunan jarak bebas kode sebesar 9% dan 33% di bawah rata-rata berturut-turut untuk laju kode 3/7 dan 3/6. Sebaliknya puncturing yang dilakukan pada bit-bit yang tersebar akan menghasilkan kinerja kode yang baik, seperti diindikasikan dengan peningkatan jarak bebas sebesar 27% dan 32.45% di atas rata-rata berturut-turut untuk laju kode 3/7 dan 3/6.

Kata kunci: *bobot galat, kode ekivalen non-punctured, kode konvolusional punctured, pola puncturing*

Abstract

Punctured convolutional codes are known to have low complexity compared to their non-punctured counterpart, while retaining a good performance. Analyzing the performance of punctured convolutional code can be simplified by using non-punctured equivalent code. In this paper new punctured convolutional codes with rates of 3/8, 3/7 and 3/6 are proposed, and their performances are studied by first constructing non-punctured equivalent codes. Simulations results show that different puncturing patterns will affect the code performances. Further investigations show that puncturing adjacent bits is to be avoided as it tends to degrade the code performance, as indicated by a decrease of the free distance by 9% and 33% below average for code rates 3/7 and 3/6 respectively. On the contrary, dispersed punctured bits will yield good code performance as indicated in the increase of the free distance by 27% and 32.45% above average for code rates 3/7 and 3/6 respectively.

Keywords: *error weight, punctured convolutional code, puncturing pattern, non-punctured equivalent code*

1. Introduction

Forward error control is one of the key areas which enable the rapid development of reliable and secure telecommunication systems. It is found in the transmission and reception parts of a telecommunication system, as well as in the storage media critical to the reliability of a system.

One of the most widely used forward error control method is convolutional coding, found in wireless terrestrial to deep space telecommunication. Its advantage lies in its ability to protect transmitted data from burst as well as intermittent errors. This type of coding plays an important role in both parallel and serial concatenated coding, which serves as a highly reliable error detector and corrector [1].

To ensure high reliability, however, convolutional codes tend to occupy a large bandwidth. This is due to the fact that convolutional codes add redundancy to each transmitted bit, producing a code rate of smaller than 1. The larger number of redundancy bits added to each transmitted bit, the stronger the protection given to the said bit against transmission errors [2].

Received March 8, 2012; Revised July 4, 2012; Accepted July 14, 2012

One way to reduce the occupied bandwidth is by using punctured convolutional codes [3-4]. Punctured convolutional codes can also provide variable rate convolutional code, an important part of unequal error protection in wireless telecommunication systems [5-6]. A puncturing process deletes a number of bits in the codeword produced by a convolutional encoder [3]. As the number of redundant bits decreases, so does the system complexity and the bandwidth required by the system. The decrease of redundant bits implies that the code performance will decrease as well. However it has been shown that there punctured codes which performances are comparable to those of convolutional codes [3], [7-8].

Quantifying the performance of punctured convolutional codes is not straightforward, and the accurate analysis using state diagram and transfer function has been widely known only for convolutional codes [1], [9-10]. A method to accurately analyze the performance of punctured convolutional codes is therefore a research topic of an urgent value. We have tried to implement a combination of the methods proposed in [1] and [9] to quantify the performance of new punctured convolutional codes using non-punctured equivalent model [10]. However the role of a puncturing pattern is not shown in [10]. In this paper we use the non-punctured equivalent model [1], [9-10] to construct a punctured convolutional code in order to quantify the performance of a punctured convolutional code, as well as demonstrating the effects of puncturing patterns used on the code performance.

This paper is organized as follows. Section 2 illustrates the basics of punctured convolutional codes. The reconstruction of equivalent non-punctured convolutional code for a punctured code is given in Section 3. Section 4 gives the simulation results and the analysis of the code performance, while the conclusion is given in Section 5.

2. Research Method

In this paper three punctured convolutional codes of rates 3/8, 3/7 and 3/6 are generated from a mothercode having a rate of 1/3. The steps required to analyze the performance of punctured convolutional codes using non-punctured equivalent codes is given in Figure 1.Details are given in the subsequent sub-sections.

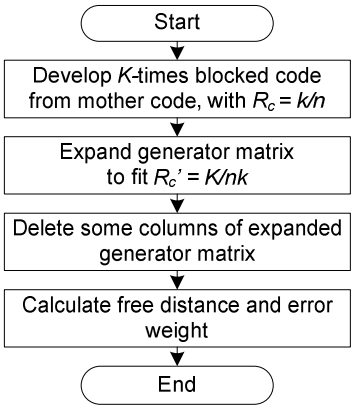

Figure 1. Steps to analyze punctured convolutional codes using non-punctured equivalent codes

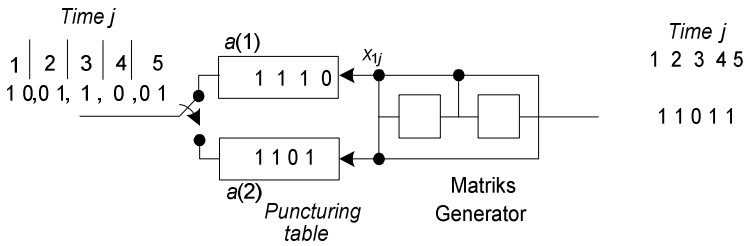

Figure 2. Punctured convolutional code with P_c=4

2.1.Punctured Convolutional Codes and the Equivalent Non-Punctured Representation

Punctured convolutional code was first developed to simplify the decoding process of a convolutional code. A pioneering research has shown that codes with rates 8/9 to 8/30, attained by puncturing a rate 8/32 convolutional code, have comparable performances to the best known convolutional codes of the respective rates [3]. Figure 2 illustrates a punctured convolutional code scheme where a convolutional code with rate $R_c=k/n=1/2$ is punctured with puncturing period $P_c=4$, where k denotes the number of transmitted bits from the source, and n denotes the number of coded bit resulted in the codeword. The illustrated system has two puncturing tables used simultaneously, where the number "0" represents a punctured bit while "1" represents a non-punctured bit. It is shown that for 4 input bits, instead of a codeword with 8 bits in accordance to $R_c=1/2$, the system yields a codeword with 6 bits, resulting in an $R_c'=4/6$. This is due to the fact that 2 bits in the codeword are punctured, conforming to the puncturing tables used.

A puncturing table can be stated as an $n\times P_c$ matrix, and for Figure 2 the puncturing table is as follows

$$a(\delta)=\begin{bmatrix}1 & 1 & 1 & 0\\ 1 & 1 & 0 & 1\end{bmatrix} \tag{1}$$

where $1 \le \delta \le (n-1)P_c$, and δ denotes the number of punctured bits. For this example as $P_c=4$, δ can take any value from 1 to 3. The resulting punctured convolutional codes are [3]

$$R_c=\frac{P_c}{P_c+\delta} \tag{2}$$

so that for the example given, the possible code rates are 4/5, 4/6 to 4/7.

It is apparent that there is a number of different puncturing tables available to reach each code rate. A different puncturing pattern will yield different code performance, as will be shown in the later sections.

To construct a non-punctured equivalent of a punctured convolutional code, the first step is to develop a K-times blocked code from the mother code with rate $R_c=1/n$, with K being any integer value. Any convolutional code with rate $1/n$ can be stated as a K-times blocked code with rate K/nK[9].

A convolutional code which generator matrix is $G=(G_0,\ldots,G_{n-1})$ can be expanded into a K-times blocked code with expanded generator G_e. The expanded generator G_e consists of n polynomials, each further broken down into K polynomials. The resulting expanded generator G_e therefore consists of nK polynomial T_{ij}, where i=0, 1, .., (n-1) and j=0,1,.., n. The expanded generator G_e for a convolutional code can be stated as [9]

$$G_e=\begin{cases} T_{j \bmod n,\left\lfloor \frac{j}{n}\right\rfloor -i} & n\times i \le j \\ DT_{j \bmod n,\left\lfloor \frac{j}{n}\right\rfloor -i+K} & n\times i > j \end{cases} \tag{3}$$

where $\lfloor j/n \rfloor$ denotes the smallest integer not exceeding j/n and D denotes the D-transformation of unit delay produced by one memory element in the shift-register of the convolutional encoder.

For the example as given in Figure 2, assume the mother code has a rate of $R_c=1/2$ and blocked 4 times, so that K=4. The resulting equivalent code rate is $K/nK=4/(2)(4)=4/8$. According to (3), the expanded generator G_e will therefore have nK=8 polynomials $T_{ij}$and DT_{ij}. After the polynomials are calculated, the elements of the expanded generator are laid out as follows [9].

$$G_e=\left[Z^{K-1}\times M \mid Z^{K-2}\times M \mid \ \ldots \mid Z\times M \mid M\ \right] \tag{4}$$

where Z denotes a $K \times K$matrix consisting of an upper diagonal of 1, a D in the bottom left corner and 0 elsewhere [9], while M is a $K \times n$ matrix which consists of the previously calculated polynomials $T_{ij}$and $DT_{ij.}$

The next step in constructing a non-punctured equivalent of a punctured convolutional code is to delete some columns in the expanded generator matrix G_e. The deletion is done according to the puncturing pattern as stated in the puncturing table.

As there are no systematic method to create a well-performing punctured convolutional code, such code is created by puncturing a well-known convolutional code with good performance [7-8].

2.2.Constructing Non-Punctured Equivalent Codes of Punctured Convolutional Codes

In this paper a mother code having a rate of 1/3 will be punctured to yield punctured convolutional codes of rates 3/8 to 3/6with puncturing period P=3. For R_c=1/3, a blocking of K=3 times is carried out, resulting in an equivalent code rate R_c'=3/9. The generator of the mother code is $G(D)$=[75 53 47], which is a well-known convolutional code with good performance. This generator will be expanded to yield G_e which consists of nK=9 polynomials. In its matrix form, G_e will have nKcolumns and n rows.

To construct G_e, first $G(D)$ is split into $G_0(D)$, $G_1(D)$, and $G_2(D)$. The three separate generators can be stated in their binary forms as follows:

$$G_0(D)=75_8=1\ 1\ 1\quad 1\ 0\ 1_2=1 + D + D^2 + D^3 + D^5 \tag{5}$$

$$G_1(D)=53_8=1\ 0\ 1\quad 0\ 1\ 1_2=1 + D^2 + D^4 + D^5 \tag{6}$$

$$G_2(D)=47_8=1\ 0\ 0\quad 1\ 1\ 1_2=1 + D^3 + D^4 + D^5 \tag{7}$$

Each of the generator $G_0(D)$, $G_1(D)$, and $G_2(D)$ is split further using the method proposed in [1]. The generator $G_0(D)$ is split as follows

$$G_0(D) = 1 + D + D^2 + D^3 + D^5=a_0 D^0+a_1 D^1+a_2D^2+a_3D^3+a_5 D^5 \tag{8}$$

where $a \overset{\Delta}{=}[a_0, a_1, \ldots, a_m]$ is binary sequence and its equivalent in the D-domain is $A(D)=\sum_{i=0}^{m} a_i D^i$ and the sequence a can be split into P sub-sequences with respect to the modulo-P positions [1]

$$\begin{aligned} a_0 &\overset{\Delta}{=}[a_0, a_t, a_{2t}, \ldots], \\ a_1 &\overset{\Delta}{=}[a_1, a_{t+1}, a_{2t+1}, \ldots], \\ &\ldots = \ldots \\ a_{t-1} &\overset{\Delta}{=}[a_{t-1}, a_{2t-1}, a_{3t-1}, \ldots] \end{aligned} \tag{9}$$

hence the representation of $G_0(D)$ in (8).

The first of the 3 polynomials T_{ij} gained from (8) is T_{00}, where

$$\begin{aligned} T_{00} &= a_{iP+j} D^i \qquad i = 0,1,2,....; j = 0 \\ &= a_{0\cdot 3+0} D^0 + a_{1\cdot 3+0} D^1 + a_{2\cdot 3+0} D^2 + a_{3\cdot 3+0} D^3 + \ldots \\ &= a_0 D^0 + a_3 D^1 + a_6 D^2 + a_9 D^3 + \ldots \end{aligned} \tag{10}$$

It is shown in (8) that the polynomials in $G_0(D)$ possesses a maximum coefficient of a_5, therefore the terms of sequence in (10) with coefficients higher than 5 are negligible. The first polynomial T_{00} can be written as

$$T_{00} = a_0 D^0 + a_3 D^1 = 1 + D \tag{11}$$

With the same method, the polynomials T_{01} and T_{02} can be derived from $G_0(D)$ and yields

$$T_{01}=1 \tag{12}$$
$$T_{02}=1+D \tag{13}$$

In a similar manner, $G_1(D)$ is rewritten as

$$G_1(D)= 1 + D^2 + D^4 + D^{5=}a_0 D^0+a_2 D^2+a_4D^4+a_5 D^5 \tag{14}$$

Splitting $G_1(D)$ into 3 polynomials will yield

$$T_{10}=1 \tag{15}$$

$$T_{11}=1+D \tag{16}$$

$$T_{12}=1+D \tag{17}$$

Whereas $G_2(D)$ is rewritten as

$$G_2(D)= 1 + D^3 + D^4 + D^{5=}a_0 D^0+a_3 D^3+a_4D^4+a_5 D^5 \tag{18}$$

and splitting $G_2(D)$ into 3 polynomials will yield

$$T_{20}=1 + D \tag{19}$$

$$T_{21}=1+D \tag{20}$$

$$T_{22}=1+D \tag{21}$$

The resulting 9 polynomials T_{00} to T_{22} are the elements of expanded matrix generator G_e, which is stated in (4). For K=3, Z can be stated as

$$Z = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ D & 0 & 0 \end{bmatrix} \tag{22}$$

while M is

$$M = \begin{bmatrix} T_{02} & T_{12} & T_{22} \\ T_{01} & T_{11} & T_{21} \\ T_{00} & T_{10} & T_{20} \end{bmatrix} \tag{23}$$

Using (4), the expanded generator G_e can be stated as

$$G_e = \begin{bmatrix} T_{00} & T_{10} & T_{20} & T_{01} & T_{11} & T_{21} & T_{02} & T_{12} & T_{22} \\ DT_{02} & DT_{12} & DT_{22} & T_{00} & T_{10} & T_{20} & T_{01} & T_{11} & T_{21} \\ DT_{01} & DT_{11} & DT_{02} & DT_{02} & DT_{12} & DT_{22} & T_{00} & T_{10} & T_{20} \end{bmatrix} \tag{24}$$

where each element $T_{ij}$is given in previous equations and DT_{ij} can be easily calculated using T_{ij} and is given in Table 1.

3. Results and Analysis

The code rates of 3/8 to 3/6 are yielded by puncturing 1 to 3 bits from the mother code used in this paper, which rate is 3/9. This translates into removing 1 to 3 columns from the expanded generator G_e. The determination of which column or columns to be removed is done by trial and error to find the best-performing punctured codes.

Table 1. Values of DT_{ij} calculated from (D), $G_1(D)$, $G_2(D)$

Polynomials from $G_0(D)$	Value (Octal)
DT_{00}	Not needed
DT_{01}	2
DT_{02}	5
DT_{10}	Not needed
DT_{11}	5
DT_{12}	5
DT_{20}	Not needed
DT_{21}	5
DT_{22}	5

Table 2. Punctured code with R_c=3/8

Code Rate: 3/8 Bit(s) Punctured: 1									
Column(s) removed	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
d_{free}	6	7	6	6	7	6	7	7	8
C_d	[2 2 3 10 18]	[2 6 7 14 47]	[2 0 10 0 25]	[2 2 0 7 17]	[4 0 4 22 22]	[2 0 10 0 31]	[4 7 13 25 18]	[2 5 11 15 10]	[7 0 29 0 101]

Removing one column of $G_e$for example will mean increasing the code rate from 3/9 to 3/8, regardless of which of the 9 columns in $G_e$is removed. However the code yielded by removing the first column in G_e will have a different performance compared to the code yielded by removing the ninth column in G_e. The performance of a punctured code resulting from removing one column in G_e is given in Table 2. It is shown that different puncturing patterns will result in different performances, in this case different free distances d_{free} and error weights c_d for the given code; where error weights represent the number of erroneous bits produced by the incorrect paths. The simulations for all codes are done for the first 5 components of c_d.

It is noted that for instance, puncturing the first bit of the codeword, which is equal to removing the first column in G_e results in a d_{free} of 6. On the contrary, puncturing the ninth bit of the codeword, will yield a d_{free} of 8 although the code rate is maintained at 3/8. This means puncturing the ninth bit is favorable to puncturing the first bit of the codeword. Puncturing the first bit will yield a code which free distance is the same as the code yielded by puncturing the third bit. However the code weight c_d will differ between the two, and the total number of bit errors in the latter code is higher than that of the code where the first bit is punctured.

The simulation result for a punctured code with R_c=3/7 is given in Table 3. To achieve R_c=3/7, 2 bits in the resulting codeword from a mother code with R_c=3/9 are punctured. This is equal to removing two columns from the expanded generator G_e. It is shown that consistent with the result shown in Table 2, different puncturing patterns will affect the code performance.

Table 3. Punctured code with R_c=3/7

Code Rate: 3/7 Bit(s) Punctured: 2								
Column(s) removed	**1 & 2**	**1 & 3**	**1 & 4**	**1 & 5**	**1 & 6**	**1 & 7**	**1 & 8**	**1& 9**
d_{free}	5	4	5	5	5	5	Catastrophic	6
C_d	[2 0 9 10 19]	[2 0 0 12 16]		[2 2 2 20 20]	[2 2 10 12 16]	[2 5 4 20 28]		[5 2 10
Column(s) removed	**2& 3**	**2&4**	**2&5**	**2&6**	**2&7**	**2&8**	**2&9**	
d_{free}	5	catastrophic	6	6	6	7	7	
C_d	[2 4 4 16 39]		[2 6 4 25 42]	[6 6 5 50 94]	[2 20 26 41 109]	[11 12 20 56 115]	[5 16 37 78 191]	
Column(s) removed	**3 & 4**	**3 & 5**	**3 & 6**	**3 & 7**	**3 & 8**	**3 & 9**		
d_{free}	5	5	6	5	6	6		
C_d	[2 2 4 13 26]	[2 0 6 10 11]	[12 0 13 0 134]	[2 4 12 19 18]	[2 10 14 10 60]	[2 0 23 0 99]		
Column(s) removed	**4 & 5**	**4 & 6**	**4 & 7**	**4 & 8**	**4 & 9**			
d_{free}	5	4	5	5	6			
C_d	[2 2 4 13 26]	[2 0 0 6 12]	[2 2 7 17 28]	[2 0 5 16 17]	[2 5 5 28 39]			
Column(s) removed	**5 & 6**	**5 & 7**	**5 & 8**	**5 & 9**				
d_{free}	5	Catastrophic	6	7				
C_d	[2 0 6 15 17]		[2 2 14 17 40]	[4 8 20 54 110]				
Column(s) removed	**6 & 7**	**6 & 8**	**6 & 9**					
d_{free}	5	5	6					
C_d	[2 4 15 24 10]	[2 0 11 15 10]	[2 0 29 0 87]					
Column(s) removed	**7 & 8**	**7 & 9**						
d_{free}	6	*Requires different constraint length*						
C_d	[5 12 14 21 73]							
Column(s) removed	**8 & 9**							
d_{free}	6							
C_d	[3 5 7 16 62]							

It is further noted that puncturing dispersed bits yield better performance, in this case larger d_{free}, compared to puncturing adjacent bits. From Table 3 it is observed that by puncturing bits 1 and 2 of a codeword, the resulting d_{free} is 5, while puncturing bits 1 and 9 will yield a d_{free} of 6. Furthermore, puncturing bits 2 and 9 will yield a d_{free} of 7 while puncturing bits 2 and 3 will result in a d_{free} of 5. The variations of d_{free} resulting from different puncturing patterns are observed in the other instances in Table 3. In all instances, it is seen that puncturing adjacent bits will yield poorer performance compared to puncturing highly-dispersed bits. This is due to the fact that puncturing adjacent bits emulates a burst error, while puncturing highly-dispersed

bits emulates an intermittent error. The convolutional decoder used in the model system of this paper performs better to mitigate intermittent error, and therefore punctured bits located more dispersedly will contribute to better performance compared to ones located next to each other. The best performances, however, are obtained by puncturing bits 2 and 8; 2 and 9; and 5 and 9. This implies that puncturing the first significant bit (bit 1) will degrade the code performance. Therefore, dispersing the punctured bits as well as keeping the first bit unpunctured will assist in keeping a good code performance.

Table 4. Punctured code with R_c=3/6

Code Rate: 3/6 Bit(s) Punctured: 3							
Column(s) removed	**1, 2, 3**	**1, 2, 4**	**1, 2, 5**	**1, 2, 6**	**1, 2, 7**	**1, 2, 8**	**1, 2, 9**
d_{free}	3	Catastrophic	4	5	4	catastrophic	5
C_d	[2 0 4 14 19]		[2 0 12 8 48]	[8 6 22 72 123]	[2 3 14 30 102]		[5 0 25 90 160]
Column(s) removed	**1, 3, 4**	**1, 3, 5**	**1, 3, 6**	**1, 3, 7**	**1, 3, 8**	**1, 3, 9**	
d_{free}	3	3	4	2	Catastrophic	4	
C_d	[2 0 2 19 25]	[2 0 2 14 20]	[2 12 0 12 108]	[2 0 4 17 28]		[2 0 8 24 82]	
Column(s) removed	**1, 4, 5**	**1, 4, 6**	**1, 4, 7**	**1, 4, 8**	**1, 4, 9**		
d_{free}	4	3	4	Catastrophic	5		
C_d	[4 0 10 42 70]	[2 0 2 19 28]	[4 3 12 18 70]		[7 10 12 53 154]		
Column(s) removed	**1, 5, 6**	**1, 5, 7**	**1, 5, 8**	**1, 5, 9**			
d_{free}	4	Catastrophic	Catastrophic	5			
C_d	[2 4 13 15 70]			[2 10 20 95 167]			
Column(s) removed	**1, 6, 7**	**1, 6, 8**	**1, 6, 9**				
d_{free}	4	Catastrophic	5				
C_d	[2 6 19 20 60]		[2 10 32 42 131]				
Column(s) removed	**1, 7, 8**	**1, 7, 9**					
d_{free}	catastrophic	*Requires different constraint length*					
C_d							
Column(s) removed	**1, 8, 9**						
d_{free}	Catastrophic						
C_d							
Column(s) removed	**2, 3, 4**	**2, 3, 5**	**2, 3, 6**	**2, 3, 7**	**2, 3, 8**	**2, 3, 9**	
d_{free}	catastrophic	4	6	4	5	5	
C_d		[2 4 4 19 37]	[6 6 5 50 94]	[2 15 23 41 129]	[6 9 25 60 132]	[2 14 29 88 244]	

Code Rate: 3/6 Bit(s) Punctured: 3							
Column(s) removed	**2, 4, 5**	**2, 4, 6**	**2, 4, 7**	**2, 4, 8**	**2, 4, 9**		
d_{free}	catastrophic	catastrophic	catastrophic	catastrophic	catastrophic		
C_d							
Column(s) removed	**2, 5, 6**	**2, 5, 7**	**2, 5, 8**	**2, 5, 9**			
d_{free}	5	catastrophic	6	6			
C_d	[6 2 17 54 88]		[8 11 36 62 252]	[2 19 49 153 368]			
Column(s) removed	**2, 6, 7**	**2, 6, 8**	**2, 6, 9**				
d_{free}	5	5	6				
C_d	[17 24 45 146 306]	[6 7 25 85 159]	[11 42 92 237 606]				
Column(s) removed	**2, 7, 8**	**2, 7, 9**					
d_{free}	5	*Requires different constraint length*					
C_d	[3 18 26 85 148]						
Column(s) removed	**2, 8, 9**						
d_{free}	5						
C_d	[3 8 16 56 181]						
Column(s) removed	**3, 4, 5**	**3, 4, 6**	**3, 4, 7**	**3, 4, 8**	**3, 4, 9**		
d_{free}	4	4	4	5	5		
C_d	[2 2 4 31 36]	[2 6 4 16 52]	[2 6 10 23 47]	[4 9 24 57 111]	[2 7 14 31 129]		
Column(s) removed	**3, 5, 6**	**3, 5, 7**	**3, 5, 8**	**3, 5, 9**			
d_{free}	5	catastrophic	5	5			
C_d	[8 6 6 49 113]		[2 6 17 36 92]	[2 0 11 62 127]			
Column(s) removed	**3, 6, 7**	**3, 6, 8**	**3, 6, 9**				
d_{free}	4	5	6				
C_d	[4 14 12 10 142	[10 8 10 75 123]	[17 0 108 0 972]				
Column(s) removed	**3, 7, 8**	**3, 7, 9**					
d_{free}	5	*Requires different constraint length*					
C_d	[12 9 21 72 157]						
Column(s) removed	**3, 8, 9**						
d_{free}	6						
C_d	[10 18 42 115 274]						

Puncturing several bits at the same time might also change the constraint length of the code. To preserve consistency, code words requiring a constant length exceeding or below [3 3 3] are not included in the simulation.

The simulation result for a punctured code with R_c=3/6 is given in Table 4. To achieve $R_{c=}$3/6, 3 bits in the resulting codeword from a mother code with $R_{c=}$3/9 are punctured. The puncturing process is equal to removing 3 columns from the expanded generator G_e of the mother code. It is observed that consistent with the previous results, different puncturing pattern will yield different code performance, suggesting that by choosing a particular puncturing pattern, the code rate can be maintained yet the code performance can be optimized.

Table 4. Punctured code with R_c=3/6 (*cont.*)

Code Rate: 3/6 Bit(s) Punctured: 3							
Column(s) removed	**4, 5, 6**	**4, 5, 7**	**4, 5, 8**	**4, 5, 9**			
d_{free}	2	catastrophic	4	5			
C_d	[2 0 0 15 32]		[2 0 6 19 48]	[27 21 52 151]			
Column(s) removed	**4, 6, 7**	**4, 6, 8**	**4, 6, 9**				
d_{free}	3	4	4				
C_d	[2 0 4 19 26]	[2 0 0 15 27]	[2 0 0 38 36]				
Column(s) removed	**4, 7, 8**	**4, 7, 9**					
d_{free}	4	*Requires different constraint length*					
C_d	[2 3 16 14 72]						
Column(s) removed	**4, 8, 9**						
d_{free}	5						
C_d	[5 8 5 42 126]						
Column(s) removed	**5, 6, 7**	**5, 6, 8**	**5, 6, 9**				
d_{free}	catastrophic	5	4				
C_d		[2 0 11 14 49]	[2 0 11 14 49]				
Column(s) removed	**5, 7, 8**	**5, 7, 9**					
d_{free}	catastrophic	*Requires different constraint length*					
C_d							
Column(s) removed	**5, 8, 9**						
d_{free}	6						
C_d	[8 7 62 84 311]						
Column(s) removed	**6, 7, 8**	**6, 7, 9**					
d_{free}	4	*Requires different constraint length*					
C_d	[2 10 17 15 64]						
Column(s) removed	**6, 8, 9**						
d_{free}	5						
C_d	[5 3 16 52 100]						
Column(s) removed	**7, 8, 9**						
d_{free}	*Requires different constraint length*						
C_d							

The best d_{free} for R_c=3/6 is 6, attainable through puncturing bits 2, 3, 6; 2, 5, 8; 2, 5, 9; 2, 6, 9; 3, 6, 9; and 3, 8, 9. This implies that both dispersing the punctured bits and keeping the most significant bit intact will contribute to improve the code performance. It is also shown that if two out of the three punctured bits are adjacent, the code can still yield good performance. This is not the case if all three punctured bits are adjacent.

For rates R_c=3/7 and R_c=3/6, there are several catastrophic codes. This is unavoidable as puncturing might change the structure of the matrix generator into a non-invertible generator polynomial. As the case with R_c=3/7, several codes with R_c=3/6 requires a constraint length other than [3 3 3] and therefore the simulation for their d_{free} is excluded.

The worst d_{free} for R_c=3/6 is 3, which is obtained by puncturing bits 1, 2, 3; 1, 3, 4; 1, 3, 5; 1, 4, 6; 4, 5, 6; and 4, 5, 7. This concurs with our previous observation that puncturing adjacent bits, especially if one of the bits happens to be the most significant bit of the mothercode, may lower the code performance. It is noticed however that puncturing bits 6, 7, 8, which are adjacent bits, still yield a d_{free} of 5. Therefore whilst there is no systematical way to determine a puncturing pattern which is beneficial to the code performance, dispersing the punctured bits is more favorable than keeping the punctured bits adjacent.

Table 5 summarizes the results in Tables 2, 3 and 4. For rate 3/8, as only one bit is punctured, only the average value of d_{free} is given. For rates 3/7 and 3/6, it is shown that puncturing adjacent bits will yield the worst d_{free}, whereas puncturing dispersed bits yields the best d_{free}.

Table 5. Summary of performance

Code rates	3/8	3/7	3/6
Average d_{free}	6.67	5.5	4.53
Worst d_{free} (obtained by puncturing adjacent bits)	Not applicable	5 (9% below average)	3 (33% below average)
Best d_{free} (obtained by puncturing dispersed bits)	as only 1 bit is punctured	7 (27% above average)	6 (32.45% above average)

4. Conclusion

New punctured convolutional codes with rates of 3/8, 3/7 and 3/6, yielded from a mother code of rate 3/9 have been proposed. Non-punctured equivalent codes are used to represent the punctured codes, to simplify the process of calculating the parameters of the code performance. These parameters are free distance and error weights.

Simulation results show that while using different puncturing patterns may retain the code rate, the code performance will vary according to the puncturing pattern used. It is shown that dispersing the punctured bits, in many cases will assist in improving the code performance, as opposed to puncturing adjacent bits. Dispersed punctured bits resemble intermittent error which a convolutional encoder/decoder is capable to mitigate. On the contrary, puncturing adjacent bits emulates a burst error which in turn degrades the encoder/decoder ability to recover lost bits. Further researches are needed to statistically predict how certain puncturing patterns affect the performance of punctured convolutional code.

References

[1] Li J, Kurtas E. *Punctured Convolutional Code Revisited: The Exact State Diagram and Its Implications*. The 38th Asilomar Conference on Signals, Systems and Computers. Pacific Grove. 2004;2: 2015-2019.
[2] Proakis JG. Digital Communications. Singapore: McGraw-Hill International. 2008.
[3] Hagenauer J. Rate-Compatible Punctured Convolutional Codes (RCPC Codes) and Their Applications. *IEEE Transaction on Communications*. 1988; 36(4): 389-400.
[4] Li J, Alqamzi H. *An Optimal Distributed and Adaptive Source Coding Strategy Using Rate-Compatible Punctured Convolutional Codes.* IEEE Conference of Acoustics, Speech and Signal Processing (ICASSP '05). Philadelphia. 2005; 5:685-688.
[5] Guo R, Zhou P, Liu J. *BER Performance Analysis of RCPC Encoded MIMO-OFDM in Nakagami-m Channels.* International Conference on Wireless Communications, Networking and Mobile Computing (WiCOM '06). Wuhan. 2006; 1-4.

[6] Noh Y, Lee H, Lee I. *Design of Unequal Error Protection for MIMO-OFDM.* IEEE 61st Vehicular Technology Conference. Stockholm. 2005; 2:1058-1062.
[7] Lee LHC. New Rate-Compatible Punctured Convolutional Codes for Viterbi Decoding. *IEEE Transactions on Communications*. 1994; COM-42: 3073-3079.
[8] Lee LHC, Sodha J. *More New Rate-Compatible Punctured Convolutional Codes for Viterbi Decoding*. Proc. 5th Workshop on Telecommunication & Signal Processing. Hobart. 2006.
[9] Cluzeau M. *Reconstruction of Punctured Convolutional Codes*. Proc. IEEE Information Theory Workshop (ITW 09). 2009: 75-79.
[10] Sari, L. Studi Penggunaan Kode Konvolusional Ekivalen Untuk Representasi Kode Konvolusional Punctured. *Jurnal Elektro Universitas Atma Jaya*. 2010; 3(2): 139-148 (in Indonesian).