J Inf Process Syst, Vol.10, No.1, pp. 69~80, March 2014
http://dx.doi.org/10.3745/JIPS.2014.10.1.069

pISSN 1976-913X
eISSN 2092-805X

The Construction and Viterbi Decoding of New (2*k*, *k*, *l*) Convolutional Codes

Wanquan Peng* and Chengchang Zhang**

Abstract—The free distance of (*n, k*, *l*) convolutional codes has some connection with the memory length, which depends on not only *l* but also on *k*. To efficiently obtain a large memory length, we have constructed a new class of (2*k*, *k*, *l*) convolutional codes by (2*k*, *k*) block codes and (2, 1, *l*) convolutional codes, and its encoder and generation function are also given in this paper. With the help of some matrix modules, we designed a single structure Viterbi decoder with a parallel capability, obtained a unified and efficient decoding model for (2*k*, *k*, *l*) convolutional codes, and then give a description of the decoding process in detail. By observing the survivor path memory in a matrix viewer, and testing the role of the max module, we implemented a simulation with (2*k*, *k*, *l*) convolutional codes. The results show that many of them are better than conventional (2, 1, *l*) convolutional codes.

Keywords—Convolutional Codes, Block Codes, Double Loop Cyclic Codes, Matrix Decoding, Viterbi Algorithm

1. INTRODUCTION

Convolutional codes have good BER performance and a memory characteristic. Early classical convolutional codes have self-orthogonal codes, orthogonalizable codes, and "quick-look-in" codes [1, 2]. In the 1970s, to search optimum codes, some scholars proposed a Viterbi decoding search algorithm by computer [3]. In the 1980s, punctured convolutional codes and tail-biting convolutional codes were widely used in a variety of digital communication systems [4]. A recursive systematic convolutional code (RSC) was developed with the invention of Turbo Codes in the 1990s [5]. An RSC is a system code with the distance characteristics of a non-systematic code. Currently, convolutional LDPC codes [6, 7] have become the new hot topic, as they can obtain a good cost performance when implementing the Belief Propagation (BP) algorithm. In addition, the quantum convolutional codes used in quantum communication have also become absorbing quantum error correcting codes [8].

In fact, the free distance of (*n*, *k*, *l*) convolutional codes depends on not only a good generator matrix, but also on more memory length *kl*. However, in the development of convolutional codes, more *l*, not *k*, are successfully increased. Based on this, by combining (2*k*, *k*) block codes with (2, 1, *l*) convolutional codes, we constructed a new class of (2*k*, *k*, *l*) convolutional codes that can have growth in both *k* and *l*. As a new kind of code, (2*k*, *k*, *l*) convolutional codes can

※ This work was supported by Natural Science Foundation Project of CQ CSTC under No.cstc2013jcyjA40055, and Foundation of Chongqing Education Commission under No.KJ122003.
Manuscript received April 5, 2013; accepted July 8, 2013.
Corresponding Author: Wanquan Peng (408502@163.com)
* Dept. College of Electrical Engineering, Chongqing Vocational Institute of Engineering, Chongqing, China (408502@163.com)
** Dept. College of Communication Engineering, Chongqing University, Chongqing, China (zcc@cqu.edu.cn)

implement the Viterbi algorithm like (2, 1, *l*) convolutional codes. In Section 3, by means of a matrix module, we designed a soft decision Viterbi matrix decoder with a parallel capability [9], and we describe the decoding process in detail. Section 4 covers the testing of the role of the max module, deals with some (2*k*, *k*, *l*) and (2, 1, *l*) convolutional codes on the Gaussian channel and BPSK, and compares the error-correcting capability of the two kinds of codes.

2. The Construction of Convolutional Codes

The encoder of new (2*k*, *k*, *l*) convolutional codes is shown in Fig.1. The input message $\boldsymbol{M}(t)$ is q *k*-bits (k>1) vector, where $\boldsymbol{M}(t)$=[$m_0(t)$ $m_1(t)$ $m_2(t)$ ...$m_{k-1}(t)$]$^{\mathrm{T}}$; i_0, i_1, i_2, …, i_l is 2^k-ary for $\boldsymbol{M}(t)$, $\boldsymbol{M}(t\text{-}1)$, …, $\boldsymbol{M}(t\text{-}l)$, respectively, that decimal scale is 0~2^k-1; $\mathbf{D}^j$ is vector register, which can delay *k*-bits information together, after *j* times delay, $\boldsymbol{M}(t\text{-}j)$=[$m_0(t\text{-}j)$ $m_1(t\text{-}j)$ $m_2(t\text{-}j)$ ... $m_{k\text{-}1}(t\text{-}j)$]$^{\mathrm{T}}$; the linear combiner 1 implements the operation according to $\sum_{j=0}^{l} g_j \boldsymbol{M}(t-j)$, and multiplies with the generation matrix $\mathbf{G}$=[$\mathbf{I}$ $\mathbf{P}^{\mathrm{T}}$]$^{\mathrm{T}}$ of (2*k*, *k*) block codes. Based on the operation of the Galois field of GF(2), we can obtain the encoded output as follows:

$$\boldsymbol{C}_1^{i_0 i_1 \dots i_l} = \begin{bmatrix} \mathbf{I} \\ \mathbf{P} \end{bmatrix} \times \sum_{j=0}^{l} g_j \boldsymbol{M}(t-j) = \begin{bmatrix} \sum_{j=0}^{l} g_j \boldsymbol{M}(t-j) \\ \mathbf{P} \sum_{j=0}^{l} g_j \boldsymbol{M}(t-j) \end{bmatrix} \tag{1}$$

Where $\mathbf{I}$ is the identity matrix, $\mathbf{I}$ and $\mathbf{P}$ are $k \times k$ matrix. At the same time, the linear combiner 2 implements the operation according to $\sum_{j=0}^{l} (g_j + h_j) \boldsymbol{M}(t-j)$. The result is input to the embedded zero module, the *k* bits 0 are embedded, then $2k \times 1$ output vector is obtained as:

$$\boldsymbol{C}_2^{i_0 i_1 \dots i_l} = \begin{bmatrix} 0 \\ \sum_{j=0}^{l} (g_j + h_j) \boldsymbol{M}(t-j) \end{bmatrix} \tag{2}$$

The final encoded output is the sum of (1) and (2), which is as follows:

$$\boldsymbol{C}^{i_0 i_1 \dots i_l} = \boldsymbol{C}_1^{i_0 i_1 \dots i_l} + \boldsymbol{C}_2^{i_0 i_1 \dots i_l} = \begin{bmatrix} \sum_{j=0}^{l} g_j \boldsymbol{M}(t-j) \\ \mathbf{P} \sum_{j=0}^{l} g_j \boldsymbol{M}(t-j) + \sum_{j=0}^{l} (g_j + h_j) \boldsymbol{M}(t-j) \end{bmatrix} \tag{3}$$

Equation (3) is named the “generation function.” In the above process, (2*k*, *k*) codes are called “embedded codes,” and can be taken from the block codes that have code rat R=1/2 and a good Hamming distance. However, even if *k* is not too large, as a result of the multiplier effect, the growth of memory length of the (2*k*, *k*, *l*) convolutional codes is more significant than those of the (2, 1, *l*) convolutional codes. So, we will pick out (6, 3) and (8, 4), which are two double loop cyclic codes, as embedded codes in [10] to construct (6, 3, *l*) and (8, 4, *l*) convolutional codes, and their generation matrix are respectively:

$$\boldsymbol{G} = \begin{bmatrix} 1\ 0\ 0\ 0\ 1\ 1 \\ 0\ 1\ 0\ 1\ 0\ 1 \\ 0\ 0\ 1\ 1\ 1\ 0 \end{bmatrix} \tag{4}$$

and:

$$\boldsymbol{G} = \begin{bmatrix} 1\ 0\ 0\ 0\ 1\ 1\ 1\ 0 \\ 0\ 1\ 0\ 0\ 0\ 1\ 1\ 1 \\ 0\ 0\ 1\ 0\ 1\ 0\ 1\ 1 \\ 0\ 0\ 0\ 1\ 1\ 1\ 0\ 1 \end{bmatrix} \tag{5}$$

We can also see that their minimum distances are respectively 3 and 4 from (4) and (5), that they have a very good length-distance cost performance, which is beneficial to the new convolutional codes. On the other hand, the polynomial coefficient of linear combiner 1 and 2 g_j, h_j can be derived from conventional (2, 1, *l*) convolutional codes that have a good free distance. For this paper, we selected the codes with optimum distance characteristics from [10], where l=1~5**,** and are shown in Tab.1, where g_0~g_l and h_0=h_l are used to delimit groups of three digits, and are expressed in octal.

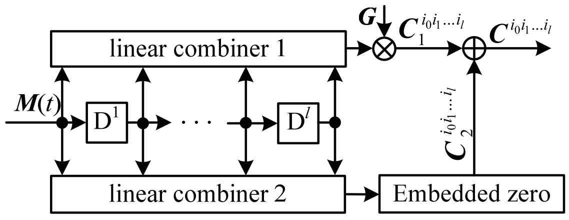

Fig. 1. Encoder of (2*k*, *k*, *l*) convolutional codes

The growth rate of the (2*k*, *k*, *l*) memory length is *k* times as that (2, 1, *l*) ones, which is very advantageous for the study of large constraint degree convolutional codes. Fig. 1 fully demonstrates the process where memory is injected from (2, 1, *l*) convolutional codes to block codes that have a similar characterize with concatenated codes [11]. However, the code rate is 1/2, which is the same as the embedded codes, and there is not a loss of concatenated codes.

Table 1. Generating polynomial coefficients of (2, 1, *l*) convolutional codes

l	$g_0 \sim g_l$	$h_0 \sim h_l$
1	3	1
2	7	5
3	17	13
4	27	31
5	75	53

3. Viterbi Matrix Decoding

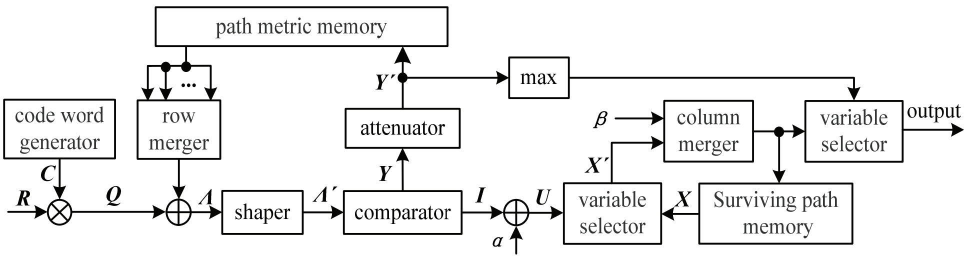

Fig. 2. Matrix decoding of (2*k*, *k*, *l*) convolutional codes

The (2*k*, *k*, *l*) convolutional codes can also decode with the Viterbi algorithm, and they can obtain faster decoding by adopting a parallel structure [12] at the expenses of the structure complexity. In fact, the trellis of (2*k*, *k*, *l*) convolutional codes has 2^{lk} states. The number of branch paths that converge at the same state node is 2^k, which can form $2^{(l+1)k}$ branches. The likelihood score of a branch path is defined as the branch metric. The accumulated value for all branch metrics of a connected path is defined as the path metric. The decoder calculates branch the metric for 2^k branches converged at one of state nodes, respectively. It also adds with the path metric from the previous time such that obtaining 2^k new path metrics, and then picks out the path with the maximum metric as a survivor path. There are 2^{lk} Add-Compare-Select operations. In order to implement the next accumulating operation, the decoder needs 2^{lk} registers to store the surviving path metric. To avoid overflow, the attenuation for the path metrics should be implemented periodically. The 2^{lk} register sets are provided for the parallel decoding to save the surviving path.

3.1 The Add-Compare-Select

With the help of some matrix modules, this paper provides a Viterbi matrix decoder with a single structure and a parallel processing capability, which is shown in Fig. 2. First, there are $2^{k(l+1)}$ branches in the trellis of (2*k*, *k*, *l*) convolutional codes. A branch has a codeword and so we can obtain a $2^{k(l+1)}\times 1$ matrix in $i_0 i_1 \ldots i_{l-1} i_l$ ascending sort by 2^k-ary:

$$\boldsymbol{C}=[\boldsymbol{C}^{00\ldots00}\ \boldsymbol{C}^{00\ldots01}\ \ldots\ \boldsymbol{C}^{00\ldots0K}\ \boldsymbol{C}^{00\ldots10}\ \ldots\ \boldsymbol{C}^{KK\ldots KK}]^{\mathrm{T}} \tag{6}$$

where $\boldsymbol{C}^{00\ldots00}\sim\boldsymbol{C}^{00\ldots0K}$ corresponds to the 2^k branches of state node $S_{00\ldots0}$, that can be derived from (3). Each element is $1\times 2k$ vector in (6), which can be converted into $2^{k(l+1)}\times 2k$ matrix:

$$\boldsymbol{C} = \begin{bmatrix} c_0^{00\cdots00} & c_0^{00\cdots01} & \cdots & c_0^{00\cdots0K} & c_0^{00\cdots10} & \cdots & c_0^{KK\cdots KK} \\ c_1^{00\cdots00} & c_1^{00\cdots01} & \cdots & c_1^{00\cdots0K} & c_1^{00\cdots10} & \cdots & c_1^{KK\cdots KK} \\ \vdots & & & \ddots & & & \vdots \\ c_{2k-1}^{00\cdots00} & c_{2k-1}^{00\cdots01} & \cdots & c_{2k-1}^{00\cdots0K} & c_{2k-1}^{00\cdots10} & \cdots & c_{2k-1}^{KK\cdots KK} \end{bmatrix}^{\mathbf{T}} \tag{7}$$

Since $\boldsymbol{C}$ is a constant matrix that can be obtained and converted to a bipolar code beforehand and can be stored in a "code word generator." We assume that the received vector with noise is $\boldsymbol{R}(t)$=[$r_0(t)$ $r_1(t)$ ... $r_{2k\text{-}1}(t)$]$^{\mathrm{T}}$. The matrix multiplier completes the "multiply" operation and the output is:

$$\boldsymbol{Q}(t) = \boldsymbol{C} \times \boldsymbol{R}(t) = \left[\rho^{00\cdots00}(t)\,\rho^{00\cdots01}(t)\cdots\rho^{00\cdots0K}(t)\,\rho^{00\cdots10}(t)\cdots\rho^{KK\cdots KK}(t)\right]^{\mathrm{T}} \tag{8}$$

where:

$$\rho^{i_0 i_1 \ldots i_{l-1} i_l}(t) = \sum_{n=0}^{2k-1} c_n^{i_0 i_1 \ldots i_{l-1} i_l} r_n(t) \tag{9}$$

Equation (9) is a inner product for $\boldsymbol{C}^{i_0 i_1 \ldots i_{l-1} i_l}$ and $\boldsymbol{R}(t)$, which is equivalent to a maximum a posteriori (MAP). Thus, the matrix multiplication between $\boldsymbol{C}$ and $\boldsymbol{R}(t)$ can complete the likelihood operation for all $2^{(l+1)k}$ branch paths. The result is in accordance with the maximum likelihood decoding. Suppose the output of the matrix adder is:

$$\boldsymbol{\Lambda}(t) = [\lambda^{00\cdots00}(t)\,\lambda^{00\cdots01}(t)\cdots\lambda^{00\cdots0K}(t)\,\lambda^{00\cdots10}(t)\cdots\lambda^{KK\cdots KK}(t)]^{\mathrm{T}} \tag{10}$$

$\boldsymbol{\Lambda}(t)$, as well as $\boldsymbol{Q}(t)$, is a $2^{(l+1)k}\times1$ matrix, in order to facilitate the implementation of the "comparison operation." The shaper then orderly extracts 2^k elements from (10), which are shaped into a new $2^{lk}\times2^k$ matrix:

$$\boldsymbol{\Lambda}'(t) = \begin{bmatrix} \lambda^{00\cdots00}(t) & \lambda^{00\cdots01}(t) & \cdots & \lambda^{00\cdots0K}(t) \\ \lambda^{00\cdots10}(t) & \lambda^{00\cdots11}(t) & \cdots & \lambda^{00\cdots1K}(t) \\ & & \vdots & \\ \lambda^{00\cdots K0}(t) & \lambda^{00\cdots K1}(t) & \cdots & \lambda^{00\cdots KK}(t) \\ & & \vdots & \\ \lambda^{KK\cdots K0}(t) & \lambda^{KK\cdots K1}(t) & \cdots & \lambda^{KK\cdots KK}(t) \end{bmatrix} \tag{11}$$

The superscript of high l bits in each column is sorted by 2^k-ary. The comparator compares 2^k elements and outputs the maximum element of each row:

$$\boldsymbol{Y}(t)=\begin{bmatrix}\max_{0\le j\le K}\left\{\lambda^{00\ldots0j}(t)\right\}\\ \max_{0\le j\le K}\left\{\lambda^{00\ldots1j}(t)\right\}\\ \vdots\\ \max_{0\le j\le K}\left\{\lambda^{00\ldots Kj}(t)\right\}\\ \vdots\\ \max_{0\le j\le K}\left\{\lambda^{KK\ldots Kj}(t)\right\}\end{bmatrix} \tag{12}$$

The role of the above process is to find the maximum likelihood path by comparing the 2^k path metrics of the state node $\mathrm{S}_{i_0 i_1\ldots i_{l-1}}$. In order to avoid the ***Y***(*t*) gradually enlarging, the attenuator finds the minimum value $\lambda_{\min}$ in (12), and subtracts the value from all rows. As such, it follows that:

$$\boldsymbol{Y}'(t)=\begin{bmatrix}\max_{0\le j\le K}\left\{\lambda^{00\ldots0j}(t)\right\}-\lambda_{\min}\\ \max_{0\le j\le K}\left\{\lambda^{00\ldots1j}(t)\right\}-\lambda_{\min}\\ \vdots\\ \max_{0\le j\le K}\left\{\lambda^{00\ldots Kj}(t)\right\}-\lambda_{\min}\\ \vdots\\ \max_{0\le j\le K}\left\{\lambda^{KK\ldots Kj}(t)\right\}-\lambda_{\min}\end{bmatrix} \tag{13}$$

Where the maximum likelihood criterion does not have to be destroyed under the same attenuation for all elements of ***Y***(*t*), and the path metric can be limited to a small value. ***Y'***(*t*) is sent to the path metric memory to be saved, due to each old state, and should point to the 2^k new states. The corresponding path metric will be called 2^k times in the next accumulation, such that 2^k times row merging is needed for ***Y'***(*t*). So, the output of the accumulator in the next time is:

$$\boldsymbol{\Lambda}(t+1)=\boldsymbol{Q}(t+1)+\begin{bmatrix}\boldsymbol{Y}'(t)\\ \boldsymbol{Y}'(t)\\ \vdots\\ \boldsymbol{Y}'(t)\end{bmatrix}=\begin{bmatrix}\rho^{00\cdots0}(t+1)+\max_{0\le j\le K}\left\{\lambda^{00\ldots0j}(t)\right\}-\lambda_{\min}\\ \rho^{00\cdots1}(t+1)+\max_{0\le j\le K}\left\{\lambda^{00\ldots1j}(t)\right\}-\lambda_{\min}\\ \vdots\\ \rho^{KK\cdots K}(t+1)+\max_{0\le j\le K}\left\{\lambda^{KK\ldots Kj}(t)\right\}-\lambda_{\min}\\ \rho^{00\cdots0}(t+1)+\max_{0\le j\le K}\left\{\lambda^{00\ldots0j}(t)\right\}-\lambda_{\min}\\ \vdots\\ \rho^{KK\cdots K}(t+1)+\max_{0\le j\le K}\left\{\lambda^{KK\ldots Kj}(t)\right\}-\lambda_{\min}\\ \vdots\\ \rho^{KK\cdots K}(t+1)+\max_{0\le j\le K}\left\{\lambda^{KK\ldots Kj}(t)\right\}-\lambda_{\min}\end{bmatrix} \tag{14}$$

The accumulator, shaper, comparator, attenuator, path metric memory, and the merger in the loop complete the add-compare-select the attenuation of the path metric and update operations.

3.2 Saving and Updating the Survivor Path

Each current state node will be connected with the surviving path that has been retained by the previous state node selected in (12). It will delete the oldest branch to complete the update of the survivor path. This procedure is completed by a variable selector and columns merger in Fig. 2. The decoder is required to provide original information groups that are consistent with the current state node $\mathrm{S}_{i_0 i_1 \ldots i_{l-1}}$. Considering that the 2^{kl} groups information vector forms the following matrices:

$$\boldsymbol{\beta}=[\boldsymbol{\beta}_{00\ldots00}\ \ \boldsymbol{\beta}_{00\ldots01}\cdots\boldsymbol{\beta}_{00\ldots0K}\ \ \boldsymbol{\beta}_{00\ldots10}\cdots\boldsymbol{\beta}_{KK\ldots KK}]^{\mathrm{T}} \tag{15}$$

where $\boldsymbol{\beta}_{i_0 i_1 \ldots i_{l-1}}=(i_0)_{\mathrm{B}}$ is $1\times k$ binary information, suppose that the current surviving path memory information is:

$$\boldsymbol{X}(t)=\begin{bmatrix} \boldsymbol{\beta}_{00\cdots00} & \boldsymbol{X}_{00\cdots00}(t-1) & \boldsymbol{X}_{00\cdots00}(t-2) & \cdots & \boldsymbol{X}_{00\cdots00}(t-\tau) \\ \boldsymbol{\beta}_{00\cdots01} & \boldsymbol{X}_{00\cdots01}(t-1) & \boldsymbol{X}_{00\cdots01}(t-2) & \cdots & \boldsymbol{X}_{00\cdots01}(t-\tau) \\ & & \vdots & & \\ \boldsymbol{\beta}_{00\cdots0K} & \boldsymbol{X}_{00\cdots0K}(t-1) & \boldsymbol{X}_{00\cdots0K}(t-2) & \cdots & \boldsymbol{X}_{00\cdots0K}(t-\tau) \\ \boldsymbol{\beta}_{00\cdots10} & \boldsymbol{X}_{00\cdots10}(t-1) & \boldsymbol{X}_{00\cdots10}(t-2) & \cdots & \boldsymbol{X}_{00\cdots10}(t-\tau) \\ & & \vdots & & \\ \boldsymbol{\beta}_{KK\cdots KK} & \boldsymbol{X}_{KK\cdots KK}(t-1) & \boldsymbol{X}_{KK\cdots KK}(t-2) & \cdots & \boldsymbol{X}_{KK\cdots KK}(t-\tau) \end{bmatrix} \tag{16}$$

This is a $2^{lk}\times k\ (\tau+1)$ matrix and the total memory depth is $k(\tau+1)$. The last k column of the matrix is the oldest group.

The role of the variable selector is to find a survivor path in the previous time that corresponds to each new branch. Let the index matrix of the maximum value in (12) be:

$$I=[i_{00\ldots00}\ \ i_{00\ldots01}\cdots i_{00\ldots0K}\ \ i_{00\ldots10}\cdots i_{KK\ldots KK}]^{\mathrm{T}} \tag{17}$$

The value range of each element is $0\sim2^k-1$. In fact, (17) is the column index of each row of (12). It needs to be transformed into the row index of (16) to be used. Let:

$$\alpha_0=\alpha_1=\ldots=\alpha_K=2^k[0\ 1\ 2\ \ldots 2^{(l-1)k}-1] \tag{18}$$

Equation (18) can be merged to obtain the correction vector:

$$\alpha=[\alpha_0\ \alpha_1\ \alpha_2\ \ldots\ \alpha_K]^{\mathrm{T}} \tag{19}$$

Equation (19) adds (17) to obtain a new row index matrix:

$$U=[u_{00\ldots00}\ \ u_{00\ldots01}\cdots u_{00\ldots0K}\ \ u_{00\ldots10}\cdots u_{KK\ldots KK}]^{\mathrm{T}} \tag{20}$$

In the next moment, the variable selector would obtain ***X***(*t*) from the surviving path memory, select new row one by one according to ***U***, and complete the reordering of all rows:

$$\boldsymbol{X}'(t)=\begin{bmatrix} \boldsymbol{\beta}_{u_{00\ldots00}} & \boldsymbol{X}_{u_{00\cdots00}}(t-1) & \boldsymbol{X}_{u_{00\cdots00}}(t-2) & \cdots & \boldsymbol{X}_{u_{00\cdots00}}(t-\tau) \\ \boldsymbol{\beta}_{u_{00\ldots01}} & \boldsymbol{X}_{u_{00\ldots01}}(t-1) & \boldsymbol{X}_{u_{00\ldots01}}(t-2) & \cdots & \boldsymbol{X}_{u_{00\ldots01}}(t-\tau) \\ & & \vdots & & \\ \boldsymbol{\beta}_{u_{00\ldots0K}} & \boldsymbol{X}_{u_{00\ldots0K}}(t-1) & \boldsymbol{X}_{u_{00\ldots0K}}(t-2) & \cdots & \boldsymbol{X}_{u_{00\ldots0K}}(t-\tau) \\ \boldsymbol{\beta}_{u_{00\ldots10}} & \boldsymbol{X}_{u_{00\ldots10}}(t-1) & \boldsymbol{X}_{u_{00\ldots10}}(t-2) & \cdots & \boldsymbol{X}_{u_{00\ldots10}}(t-\tau) \\ & & \vdots & & \\ \boldsymbol{\beta}_{u_{KK\ldots KK}} & \boldsymbol{X}_{u_{KK\ldots KK}}(t-1) & \boldsymbol{X}_{u_{KK\ldots KK}}(t-2) & \cdots & \boldsymbol{X}_{u_{KK\ldots KK}}(t-\tau) \end{bmatrix} \tag{21}$$

Then, merge the column with *β* and automatically delete the last *k* columns to obtain a new survivor path matrix:

$$\boldsymbol{X}'(t)=\begin{bmatrix} \boldsymbol{\beta}_{00\ldots00} & \boldsymbol{\beta}_{u_{00\ldots00}} & \boldsymbol{X}_{u_{00\cdots00}}(t-1) & \cdots & \boldsymbol{X}_{u_{00\cdots00}}(t-\tau+1) \\ \boldsymbol{\beta}_{00\ldots01} & \boldsymbol{\beta}_{u_{00\ldots01}} & \boldsymbol{X}_{u_{00\ldots01}}(t-1) & \cdots & \boldsymbol{X}_{u_{00\ldots01}}(t-\tau+1) \\ & & \vdots & & \\ \boldsymbol{\beta}_{00\ldots0K} & \boldsymbol{\beta}_{u_{00\ldots0K}} & \boldsymbol{X}_{u_{00\ldots0K}}(t-1) & \cdots & \boldsymbol{X}_{u_{00\ldots0K}}(t-\tau+1) \\ \boldsymbol{\beta}_{00\ldots10} & \boldsymbol{\beta}_{u_{00\ldots10}} & \boldsymbol{X}_{u_{00\ldots10}}(t-1) & \cdots & \boldsymbol{X}_{u_{00\ldots10}}(t-\tau+1) \\ & & \vdots & & \\ \boldsymbol{\beta}_{KK\ldots KK} & \boldsymbol{\beta}_{u_{KK\ldots KK}} & \boldsymbol{X}_{u_{KK\ldots KK}}(t-1) & \cdots & \boldsymbol{X}_{u_{KK\ldots KK}}(t-\tau+1) \end{bmatrix} \tag{22}$$

With the increase of the memory depth, some survivor paths will be brought together, which is reflected in the right column where the element will gradually become more identical. The role of the max module is that it outputs the index of the largest element in ***Y***′(*t*). The variable selector selects the last *k* columns of (22) as the output based on the index, which can effectively reduce memory depth. This is because even if the last node survivor path does not completely meet together, the decoder can still choose the best path as the decoding output.

Matrix processing makes the (2*k* *k*, *l*) convolutional codes decoder have the same single structure, and it achieves a high unity for the viterbi algorithm. It only needs to modify the inner parameters of some modules with different types of codes. This is very conducive to analysis and design.

4. Simulation Analysis

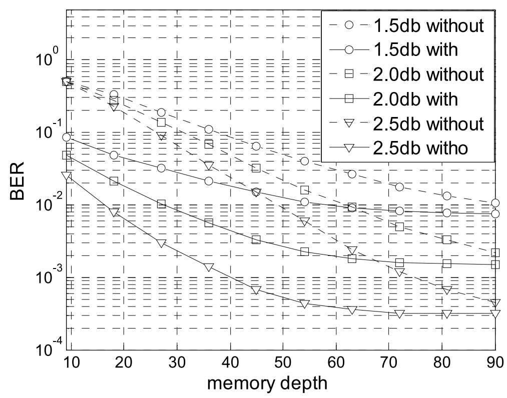

Fig. 3. Convergence of BER performance for memory depth

Simulation is implemented in the Gaussian channel and BPSK modulation. Double precision data is used in add-compare-select. It is well known that if the memory depth is too small it will affect the error-correcting capability and an excessive increase in the spending of the decoder. The first task is to verify the validity of the max module, and to determine a suitable memory depth. Take the (6, 3, 3) convolutional code as an example where 10 different memory depths are selected based on integer times of *kl*=9. When the Eb/No is respectively1.5db, 2db, and 2.5db, the BER is shown with or without the max module separately in Fig. 3. The results indicate that the convergence of the decoder was significantly improved after the max module was introduced, and it is obvious that the max module is really able to reduce the memory cells. When the memory depth is approximately 6×*kl*=54 the decoder will approach the best performance. Therefore, the memory depth will be 6*kl* in the simulation listed below.

In order to monitor the decoding process, we can observe the output of the survivor path

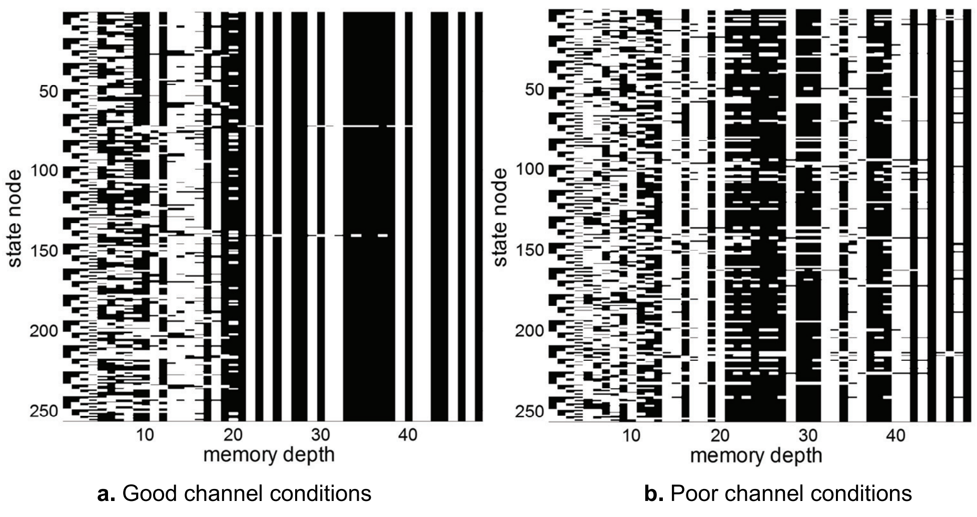

Fig. 4. Screenshots of the path memory matrix for (8, 4, 2) convolutional codes

memory with a matrix viewer. Fig. 4 shows the screen capture of (8, 4, 2) convolutional codes, where the memory depth τ is set to $6\times kl$=48, the numbers for the state node are 2^{kl}=256, and the black and white denote 1 and 0. By comparing the two figures, we can implement a helpful observation for a real time channel condition.

Combining Tab. 1 with (4) and (5), we construct five (6, 3, *l*) convolutional codes and four (8, 4, *l*) convolutional codes. Their BER performances are shown in the real line of Fig. 5 and Fig. 6. It can be seen that the SNR can obtain a stable gain with the growth of *l*. For example, when only *l*=4, the SNR is 2.9 and 2.4dB at BER=10^{-5}.

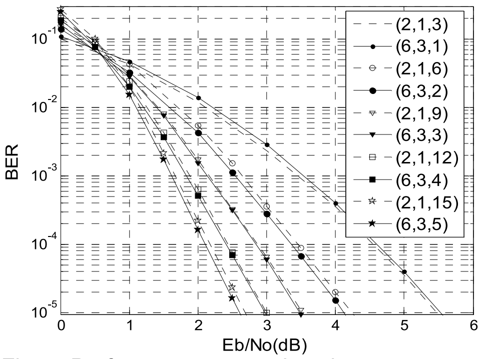

Fig. 5. Performance comparison between (6, 3, *l*) and (2, 1, *l*) convolutional codes

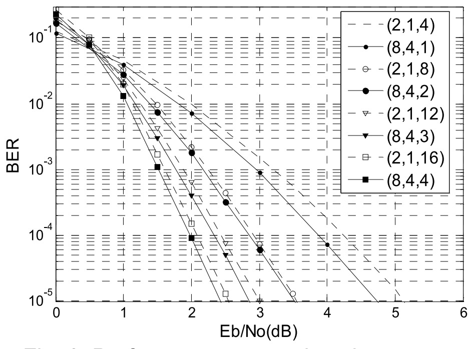

Fig. 6. Performance comparison between (8, 4, *l*) and (2, 1, *l*) convolutional codes

In order to analyze the BER performance of (2*k*, *k*, *l*) convolutional codes further, we will compare it with (2, 1, *l*) conventional convolutional codes. Let $l_1$and l_2 be the encoding restriction of (2*k*, *k*, *l*) and (2, 1, *l*) convolutional codes respectively. When $k\times l_1=l_{2,}$ their memory length and state number are equal. The decoding complexity is generally the same, so it is comparable between the two codes. The best eight (2, 1, *l*) convolutional codes derived from [10] are listed in Table 2, and their performances are added respectively as the dotted line of Fig. 5 and Fig. 6. It can be seen that, except for (6, 3, 1), the other (2*k*, *k*, *l*) convolutional codes have different degrees of advantage over (2, 1, *l*) convolutional codes.

Table 2. Generated polynomial coefficients of (2, 1, *l*) convolutional codes

L	$g_0\sim g_l$	$h_0\sim h_l$
3	17	13
4	27	31
6	117	155
8	435	657
9	1671	1233
12	15521	10677
15	152711	126723
16	205347	375145

5. Conclusion

The method to construct long codes with short ones has always been the main point of error correcting codes. In this paper, we showed how we constructed a new class of $(2k, k, l)$ convolutional codes with earlier $(2k, k)$ block codes, which visibly hold this feature. The $(2k, k, l)$ convolutional codes can achieve k-times increase in the memory length. We also developed a new approach for the study of large memory convolutional codes. For Viterbi decoding, we proposed a single structured decoder with a parallel processing capability by introducing a series of matrix modules, and we ended up with a high quality of decoding model. However, the complexity of the Viterbi algorithm is the exponential growth of kl, which results in its advantage not being able to be fully shown. We will keep this problem as we carry out new research to discover the existence of the relationship between the $(2k, k, l)$ convolutional codes and LDPC, and will fully uncover its potential error-correcting capability.

References

[1] Robinson J, Bernstein A. A class of binary recurrent codes with limited error propagation[J]. IEEE Transactions on Informational Theory, vol.13, 1967, pp.106-113.

[2] Massey J, Costello D. Nonsystematic Convolutional Codes for Sequential Decoding in Space Applications[J]. IEEE Transactions on Communication Technology, vol.19, 1971, pp.806-813.

[3] Bahl L, et al. An efficient algorithm for computing free distance[J]. IEEE Transactions on Informational Theory, vol.18, 1972, pp.437 - 439.

[4] Shu Lin, Daniel J, Costello. Error Control Coding: Fundamentals and Applications [M]. 2nd ed. Pearson Education, 2004, pp.582-598.

[5] Berrou C, Glavieux A. Near optimum error correcting coding and decoding: turbo-codes[J]. IEEE Transactions on Communication Theory, vol.44, 1996, pp.1261-1271.

[6] Pusane A E, et al. Deriving Good LDPC Convolutional Codes from LDPC Block Codes[J]. IEEE Transactions on Informational Theory, vol.57, 2011, pp. 835-857.

[7] Iyengar A R, Papaleo M, et al. Windowed Decoding of Protograph-based LDPC Convolutional Codes over Erasure Channels[J]. IEEE Transactions on Informational Theory, vol.58, 2012, pp.2303-2320.

[8] Houshmand M, Hosseini-Khayat S, Wilde M M. Minimal-Memory, Noncatastrophic, Polynomial-Depth Quantum Convolutional Encoders[J]. IEEE Transactions on Informational Theory, vol.59, 2013, pp.1198-1210.

[9] Jie Luo. On Low-Complexity Maximum-Likelihood Decoding of Convolutional Codes[J]. IEEE Transactions on Informational Theory, vol.54, 2008, pp.5756-5760.

[10] Wang Xinmei. Error Correcting Codes-Princip1e and Method[M]. Xi'an: Xidian University Press, 2001, pp.159-161,455. (in Chinese)

[11] Huebner A, Kliewer J, Costello D J. Double Serially Concatenated Convolutional Codes With Jointly Designed S-Type Permutors [J]. IEEE Transactions on Informational Theory, vol.55, 2009, pp.5811-5821.

[12] Jie Jin, Chi-ying Tsui. Low-Power Limited-Search Parallel State Viterbi Decoder Implementation Based on Scarce State Transition[J]. IEEE Transactions on VLSI Systems, vol.15, 2007, pp.1172-1176.

Wanquan Peng

He received his M.S. degrees in Circuits and Systems from Chongqing University, Chongqing, China in 2005. He is working in Chongqing Vocational Institute of Engineering. His research interests include error control coding and information theory. In the period 2004-2009 his interests was in the algorithm research and hardware design for product code. Currently he is working with new convolutional codes and LDPC convolutional codes.

Chengchang Zhang

He received the M.S. degree in Communication and information systems in 2005 and the Ph.D. degree in 2011, from the Chongqing University. He is working in Chongqing University. In 1998, he joined the Chinese Institute of Electronics, CIE. His main research interests include error correction coding, software radio and Multi-FPGA systems design.