



Viterbi Algorithm for Decoding of Convolutional Codes

Viterbi algorithm is a well-known maximum-likelihood algorithm for decoding of convolutional codes. In this article this algorithm is described using the approach suitable both for hardware and software implementations.

Introduction

Convolutional codes are frequently used to correct errors in noisy channels. They have rather good correcting capability and perform well even on very bad channels (with error probabilities of about 10^{-3}). Convolutional codes are extensively used in satellite communications.

Although convolutional encoding is a simple procedure, decoding of a convolutional code is much more complex task. Several classes of algorithms exist for this purpose:

1. *Threshold decoding* is the simplest of them, but it can be successfully applied only to the specific classes of convolutional codes. It is also far from optimal.
2. *Sequential decoding* is a class of algorithms performing much better than threshold algorithms. Their serious advantage is that decoding complexity is virtually independent from the length of the particular code. Although sequential algorithms are also suboptimal, they are successfully used with very long codes, where no other algorithm can be acceptable. The main drawback of sequential decoding is unpredictable decoding latency.
3. *Viterbi decoding* is an optimal (in a *maximum-likelihood* sense) algorithm for decoding of a convolutional code. Its main drawback is that the decoding complexity grows exponentially with the code length. So, it can be utilized only for relatively short codes.

There are also *soft-output* algorithms, like *SOVA (Soft Output Viterbi Algorithm)* or *MAP algorithm*, which provide not only a decision, but also an estimate of its reliability. They are used primarily in the decoders of *turbo codes* and are not discussed in this article.

Convolutional Codes

Convolutional Encoders

Like any error-correcting code, a convolutional code works by adding some structured redundant information to the user's data and then correcting errors using this information.

A convolutional encoder is a *linear system*.

A binary convolutional encoder can be represented as a *shift register*. The outputs of the encoder are modulo 2 sums of the values in the certain register's cells. The input to the encoder is either the unencoded sequence (for *non-recursive codes*) or the unencoded sequence added with the values of

some register's cells (for *recursive codes*).

Convolutional codes can be *systematic* and *non-systematic*. Systematic codes are those where an unencoded sequence is a part of the output sequence. Systematic codes are almost always recursive, conversely, non-recursive codes are almost always non-systematic.

A combination of register's cells that forms one of the output streams (or that is added with the input stream for recursive codes) is defined by a *polynomial*. Let m be the maximum degree of the polynomials constituting a code, then $K = m + 1$ is a *constraint length* of the code.

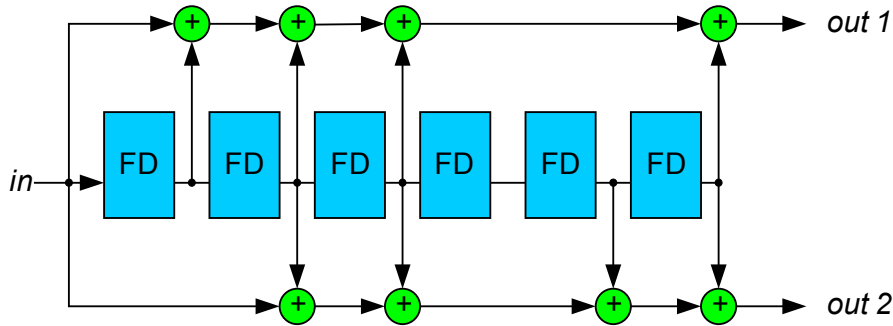


Fig. 1: A standard NASA convolutional encoder with polynomials (171,133)

For example, for the decoder on the Figure 1, the polynomials are:

- $g_1(z) = 1 + z + z^2 + z^3 + z^6$,
- $g_2(z) = 1 + z^2 + z^3 + z^5 + z^6$.

A code rate is an inverse number of output polynomials.

For the sake of clarity, in this article we will restrict ourselves to the codes with rate $R = \frac{1}{2}$.

Decoding procedure for other codes is similar.

Encoder polynomials are usually denoted in the octal notation. For the above example, these designations are “1111001” = 171 and “1011011” = 133.

The constraint length of this code is 7.

An example of a recursive convolutional encoder is on the Figure 2.

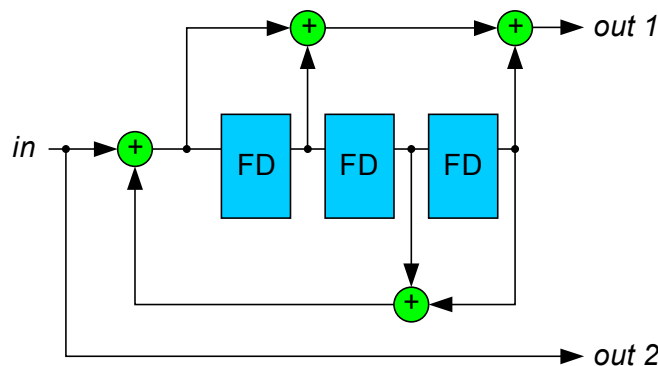


Fig. 2: A recursive convolutional encoder

Trellis Diagram

A convolutional encoder is often seen as a *finite state machine*. Each state corresponds to some value of the encoder's register. Given the input bit value, from a certain state the encoder can move to two other states. These state transitions constitute a diagram which is called a *trellis diagram*.

A trellis diagram for the code on the Figure 2 is depicted on the Figure 3. A solid line corresponds to input 0, a dotted line – to input 1 (note that encoder states are designated in such a way that the rightmost bit is the newest one).

Each path on the trellis diagram corresponds to a valid sequence from the encoder's output. Conversely, any valid sequence from the encoder's output can be represented as a path on the trellis diagram. One of the possible paths is denoted as red (as an example).

Note that each state transition on the diagram corresponds to a pair of output bits. There are only two allowed transitions for every state, so there are two allowed pairs of output bits, and the two other pairs are forbidden. If an error occurs, it is very likely that the receiver will get a set of forbidden pairs, which don't constitute a path on the trellis diagram. So, the task of the decoder is to find a path on the trellis diagram which is the closest match to the received sequence.

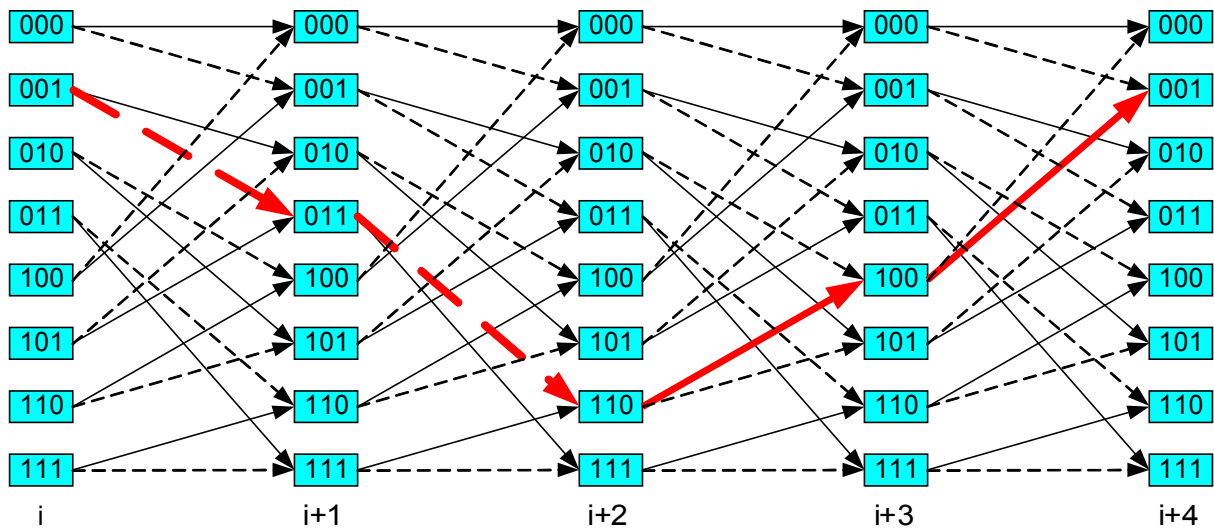


Fig. 3: A trellis diagram corresponding to the encoder on the Figure 2

Let's define a *free distance* d_f as a minimal Hamming distance between two different allowed binary sequences (a Hamming distance is defined as a number of differing bits).

A free distance is an important property of the convolutional code. It influences a number of closely located errors the decoder is able to correct.

Viterbi Algorithm

We will use the code generated by the encoder on Figure 2 as an example.

Basic Definitions

Ideally, Viterbi algorithm reconstructs the maximum-likelihood path given the input sequence.

Let's define some terms:

A *soft decision decoder* – a decoder receiving bits from the channel with some kind of reliability estimate. Three bits are usually sufficient for this task. Further increasing soft decision width will

increase performance only slightly while considerably increasing computational difficulty. For example, if we use a 3-bit soft decision, then “000” is the strongest *zero*, “011” is a weakest *zero*, “100” is a weakest *one* and “111” is a strongest *one*.

A *hard decision decoder* – a decoder which receives only bits from the channel (without any reliability estimate).

A *branch metric* – a distance between the received pair of bits and one of the “ideal” pairs (“00”, “01”, “10”, “11”).

A *path metric* – a sum of metrics of all branches in the path.

A meaning of *distance* in this context depends on the type of the decoder:

- for a *hard decision* decoder it is a *Hamming distance*, i.e. a number of differing bits;
- for a *soft decision* decoder it is an *Euclidean distance*.

In these terms, the maximum-likelihood path is a path with the minimal path metric. Thus the problem of decoding is equivalent to the problem of finding such a path.

Let's suppose that for every possible encoder state we know a path with minimum metric ending in this state. For any given encoder state there is two (and only two) states from which the encoder can move to that state, and for both of these transitions we know branch metrics. So, there are only two paths ending in any given state on the next step. One of them has lesser metric, it is a *survivor path*. The other path is dropped as less likely. Thus we know a path with minimum metric on the next step, and the above procedure can be repeated.

Implementation

A Viterbi algorithm consists of the following three major parts:

1. *Branch metric calculation* – calculation of a distance between the input pair of bits and the four possible “ideal” pairs (“00”, “01”, “10”, “11”).
2. *Path metric calculation* – for every encoder state, calculate a metric for the survivor path ending in this state (a survivor path is a path with the minimum metric).
3. *Traceback* – this step is necessary for hardware implementations that don't store full information about the survivor paths, but store only one bit decision every time when one survivor path is selected from the two.

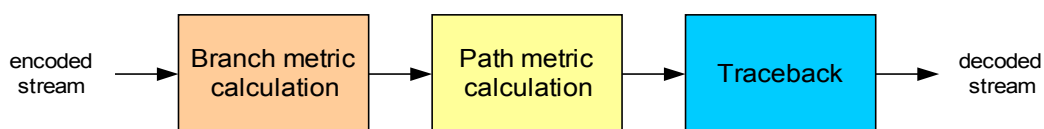


Fig. 4: Viterbi decoder data flow

Branch Metric Calculation

Methods of branch metric calculation are different for hard decision and soft decision decoders.

For a *hard decision* decoder, a branch metric is a Hamming distance between the received pair of bits and the “ideal” pair. Therefore, a branch metric can take values of 0, 1 and 2. Thus for every input pair we have 4 branch metrics (one for each pair of “ideal” values).

For a *soft decision* decoder, a branch metric is measured using the Euclidean distance. Let x be the first received bit in the pair, y – the second, x_0 and y_0 – the “ideal” values. Then branch metric is

$$M_b = (x - x_0)^2 + (y - y_0)^2 .$$

Furthermore, when we calculate 4 branch metric for a soft decision decoder, we don't actually need to know absolute metric values – only the difference between them makes sense. So, nothing will change if we subtract one value from the all four branch metrics:

$$M_b = (x^2 - 2x x_0 + x_0^2) + (y^2 - 2y y_0 + y_0^2) ;$$

$$M_b^* = M_b - x^2 - y^2 = (x_0^2 - 2x x_0) + (y_0^2 - 2y y_0) .$$

Note that the second formula, M_b^* , can be calculated without hardware multiplication: x_0^2 and y_0^2 can be pre-calculated, and multiplication of x by x_0 and y by y_0 can be done very easily in hardware given that x_0 and y_0 are constants.

It should be also noted that M_b^* is a signed variable and should be calculated in 2's complement format.

Path Metric Calculation

Path metrics are calculated using a procedure called *ACS (Add-Compare-Select)*. This procedure is repeated for every encoder state.

1. *Add* – for a given state, we know two states on the previous step which can move to this state, and the output bit pairs that correspond to these transitions. To calculate new path metrics, we add the previous path metrics with the corresponding branch metrics.
2. *Compare, select* – we now have two paths, ending in a given state. One of them (with greater metric) is dropped.

As there are 2^{K-1} encoder states, we have 2^{K-1} survivor paths at any given time.

It is important that the difference between two survivor path metrics cannot exceed $\delta \log(K-1)$, where δ is a difference between maximum and minimum possible branch metrics.

The problem with path metrics is that they tend to grow constantly and will eventually overflow. But, since the absolute values of path metric don't actually matter, and the difference between them is limited, a data type with a certain number of bits will be sufficient.

There are two ways of dealing with this problem:

1. Since the absolute values of path metric don't actually matter, we can at any time subtract an identical value from the metric of every path. It is usually done when *all* path metrics exceed a chosen threshold (in this case the threshold value is subtracted from every path metric). This method is simple, but not very efficient when implemented in hardware.
2. The second approach allows overflow, but uses a sufficient number of bits to be able to detect whether the overflow took place or not. The *compare* procedure must be modified in this case.



Fig. 5: A modulo-normalization approach

The whole range of the data type's capacity is divided into 4 equal parts. If one path metric is in the 3-rd quarter, and the other – in the 0-th, then the overflow took place and the path in

the 3-rd quarter should be selected. In other cases an ordinary compare procedure is applied. This works, because a difference between path metrics can't exceed a threshold value, and the range of path variable is selected such that it is at least two times greater than the threshold.

Traceback

It has been proven that all survivor paths merge after decoding a sufficiently large block of data (D on Figure 5), i.e. they differ only in their endings and have the common beginning.

If we decode a continuous stream of data, we want our decoder to have finite latency. It is obvious that when some part of path at the beginning of the graph belongs to every survivor path, the decoded bits corresponding to this part can be sent to the output. Given the above statement, we can perform the decoding as follows:

1. Find the survivor paths for $N+D$ input pairs of bits.
2. *Trace back* from the end of any survivor paths to the beginning.
3. Send N bits to the output.
4. Find the survivor paths for another N pairs of input bits.
5. Go to step 2.

In these procedure D is an important parameter called *decoding depth*. A decoding depth should be considerably large for quality decoding, no less then $5K$. Increasing D decreases the probability of a decoding error, but also increases latency.

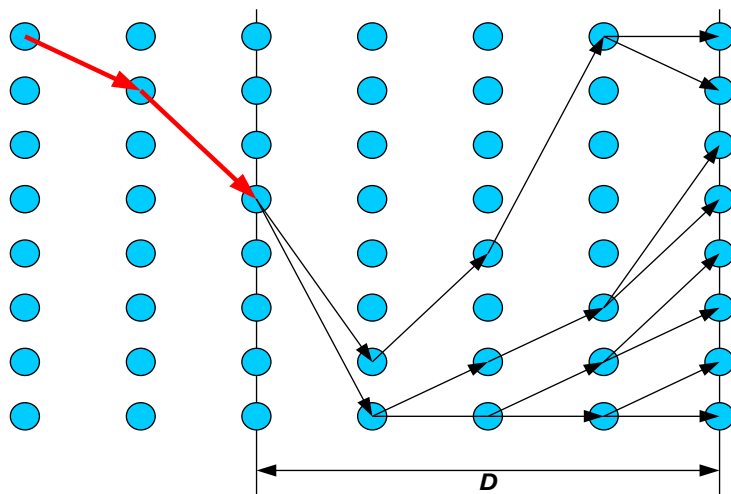


Fig. 6: Survivor paths graph example. Blue circles denote encoder states. It can be seen that all survivor paths have a common beginning (red) and differ only in their endings.

As for N , it specifies how many bits we are sending to the output after each traceback. For example, if $N=1$, the latency is minimal, but the decoder needs to trace the whole tree every step. It is computationally ineffective. In hardware implementations N usually equals D .

Advanced Topics

Punctured Codes

Punctured convolutional codes are derived from ordinary codes by dropping some output bits (i.e. not sending them to a channel). The resulting code has higher rate (less redundancy) but lower correcting capability.

The order of dropping is defined by a *puncturing matrix*. The number of rows in a puncturing matrix is equal to the number of output polynomials. The elements of this matrix are 1's (which means that we transmit a given bit) and 0's (which means that we drop it). The puncturing matrix is applied to the output stream cyclically.

The standard puncturing matrices for the rate $\frac{1}{2}$ convolutional code are:

Rate	Puncturing Matrix
$\frac{2}{3}$	$P = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$
$\frac{3}{4}$	$P = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$
$\frac{5}{6}$	$P = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix}$
$\frac{7}{8}$	$P = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$

For example, consider a puncturing matrix for the rate $\frac{2}{3}$. It means that we will transmit only every second output bit from the first polynomial, but every bit from the second polynomial.

In order to decode a punctured convolutional code, one need first to insert *erasure marks* to the places where deleted bits should be. This task is performed by a *depuncturer*. Then, the Viterbi decoder should ignore the erased bits during branch metric calculation.

Quantization

Using soft decision decoding is recommended for Viterbi decoders, since it can give a gain of about 2 dB (that is, a system with a soft decision decoder can use 2 dB less transmitting power than a system with a hard decision decoder with the same error probability).

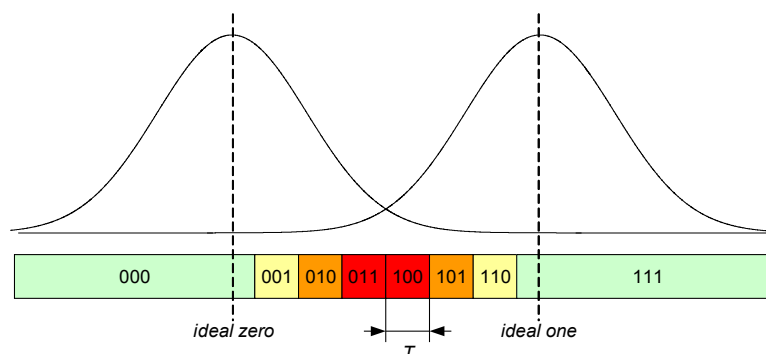


Fig. 7: Quantization zones for a 3-bit soft decision

In order to fully exploit the advantages of soft decision decoding, the signal must be properly *quantized*. The optimal quantization thresholds depends upon a noise spectral power density.

Quantization zone width (T on Figure 7) is calculated using the following formula:

$$T = \sqrt{\frac{N_0}{2^k}},$$

where k is a soft decision width (in bits).

Automatic gain control circuit usually sets ideal points on basis of total energy (signal+noise), not only noise power. Therefore, some table-based rescaling is needed to achieve maximum performance.

Conclusion

Using convolutional codes together with Viterbi algorithm can provide a large coding gain and good performance (a fully parallel Viterbi decoder implemented in hardware is able to process one input pair of bits per clock).

It should be noted that errors that Viterbi decoder can't fix appear at the decoder output in “bursts”, short erroneous blocks. For that reason, convolutional codes are often used in pair with Reed-Solomon codes.



Fig. 8: A concatenated code with inner convolutional code and outer Reed-Solomon code

Data are encoded firstly with a Reed-Solomon code, then they are processed by an interleaver (which places symbols from the same Reed-Solomon codeword far from each other), and then encoded with a convolutional code.

At the receiver, data are firstly processed by a Viterbi decoder. The bursts of errors from its output are then deinterleaved (with erroneous symbols from one burst getting into different Reed-Solomon codewords) and decoded with a Reed-Solomon decoder.

1-CORE Technologies is a leading Russian electronics design company specializing primarily in the fields of digital communications, digital signal processing and automatic control.
We have been developing commercial FEC coders and decoders since 2004.

Address: 24 Radio str., Moscow, Russia, 105005

Tel.: +7 (495) 921-79-91

Web: <http://www.1-core.com>

E-mail: sales@1-core.com