EE 435/535: Error Correcting Codes
Project 2, Fall 2009:
Viterbi Decoding of Convolutional Codes

Project #2 is due on Tuesday, November 10, 2009, in class.

1 Introduction

Project #2 is to be done individually, although student interaction is encouraged. The project report must be done individually, and may be handed in as hard copy or emailed in electronic form to the instructor.

Undergraduates should complete sections 2 through 5. Graduate students should also complete sections 2 through 5, plus section 6, their choice of problem (section 6 offers two alternatives). Undergraduates may also do a portion of section 6 for extra credit. There are also three bonus problems in section 5. Both undergraduates and graduate students may do any or all of the bonus problems for extra credit, but it is not required for either undergraduates or graduate students. Note: Graduate students may do both problems presented in section 6, rather than only one of the two, for extra credit, if they choose.

1.1 Project Reports

Show all your work! Simply giving the numeric answer to a question is not sufficient to receive full credit.

If you use Matlab to solve any of the project questions, please include your Matlab code as well as any requested data plots in your report. Similarly, if you use C/C++ or VHDL, please include your C/C++ or VHDL code. You can include the code as an appendix or as a separate file in your email. Also include all input used and output obtained from your code in answer to any project questions.

If you are unable to solve a project problem, or your code does not work, please include whatever work you have done in your project report so that I can give you partial credit for the work you did do.

2 Convolutional Codes: Encoding

Exercise:

1. Encoder: Write a program in either Matlab, C/C++ or VHDL, which convolutionally encodes 1 input bit stream $u$ into 2 output bit streams $v^{(1)}$ and $v^{(2)}$, as determined by the encoder generator sequences $g^{(1)} = [1 0 1]$ and $g^{(2)} = [1 1 1]$. Each input bit stream should have length $L$.

(a) Your program input parameters should include $L$ and $G$, the matrix of the encoder generator sequences.

(b) Your program should first generate an information bit stream $u$, as a random sequence of $L$ bits.

(c) Your program should calculate the 2 encoded bit streams $v^{(1)}$ and $v^{(2)}$, given an information bit stream $u$.

(d) Your program should add $m$ tail bits to terminate the codeword in the zero state. Feedforward encoders are forced to the all-zeros state by using zeros as the tail bits. Note that $m$ is the size of the encoder memory.
(e) Your program output should be the encoded output bit streams $v^{(1)}$ and $v^{(2)}$. They should be interleaved together so that the output codeword is
$$v = [v_1^{(1)} v_1^{(2)} v_2^{(1)} v_2^{(2)} \ldots v_{L+m}^{(1)} v_{L+m}^{(2)}].$$

(f) One method of programming the encoding process is to use the state transition table to generate the current output codeword bits, based on the current input bit and the current state, then update the state. This method requires that you generate the state transition table. Since you will need the state transition table anyway for decoding, this is no additional work. You may either generate the state transition table by hand (in which case, include it in your report) or write a function which generates the table for you (include both your function and its output, the state transition table in your report).

Note: You are not required to use this method to encode your input. You may instead build a linear feedforward shift register for your encoder, or any other method that generates the correct codeword bits.

2. **Modulation:** Rather than transmitting the codeword $v$ as bits 0 or 1, modulate the codeword $v$ into a sequence $x$ such that $v_i = 0$ is transmitted as $x_i = -1$ and $v_i = 1$ is transmitted as $x_i = 1$, then transmit the modulated sequence $x$. This modulation scheme is known as both BPSK (binary phase-shift-keying) and 2-PAM (pulse amplitude modulation). The modulated codeword is $x = 2v - 1$, and $x$ is your transmitted sequence. Write a function (or a line in your previous encoder function) that converts your codeword $v$ to a modulated sequence $x$ which will be transmitted.

3. Include your encoder and modulation function in your report. Please add some comments to enhance readability.

3 **AWGN Channel Simulation**

Exercise:

1. Write an additive white Gaussian noise (AWGN) simulator, as per project 1. If you already wrote one for project 1, you may reuse it here. Your AWGN simulator should take as input the desired variance $\sigma_n^2$ and the required length of your noise sequence, and output a noise sequence with variance $\sigma_n^2$ of the correct length.

2. Add a noise sequence $n$ of length $2(L+m)$ to your transmitted sequence $x$, to generate your received noisy modulated coded sequence $y = x + n$.

3. Include your noise generation program in your report.

4 **Viterbi Decoding Algorithm**

The Viterbi decoding algorithm is a maximum-likelihood (ML) sequence estimator, which estimates the most probable codeword sequence $v$ (corresponding to the most probable modulated codeword sequence $x$, given the received noisy codeword $y$. The Viterbi algorithm applies ML decoding to a time-invariant state trellis, one trellis section per symbol time. Each path through the entire trellis corresponds to a unique codeword. A key feature of the Viterbi decoding algorithm is that it reduces complexity by eliminating
paths that are non-contenders for the maximum-likelihood codeword; thus it does not have to evaluate all possible codewords in making its ML decision.

As in project 1, the maximum-likelihood decision is given by (for equally-likely codewords $v$)

$$\hat{v} = \max_{V_j(X_j)} p(y|x = X_j).$$

(1)

where $V_j$ indicates the codeword associated with the transmitted modulated sequence $X_j$ which maximizes $p(y|x)$. Assuming i.i.d. noise samples $n_i$, we find $p(y|x)$ as the product of the individual probabilities $p(y_i|x_i)$. As there are 2 codeword bits produced every symbol time for a rate 1/2 code, the total probability of $y$ given a particular codeword $X_j$ is found as

$$p(y|x = X_j) = \prod_{i=1}^{L+m} \prod_{c=1}^2 p(y_i^{(c)}|x_i^{(c)}),$$

(2)

for an input sequence of length $L + m$ (including tail bits) by taking the product of both conditional probabilities $p(y_i^{(1)}|x_i^{(1)})p(y_i^{(2)}|x_i^{(2)})$ at each time step $i$, for $i = 1, \ldots L + m$.

The ML codeword is then found by choosing $\hat{v}$ to correspond with the $X_j$ that maximizes Equation 2.

The individual conditional probabilities $p(y_i^{(c)}|x_i^{(c)})$ are found from the Gaussian noise distribution, because $p(y_i^{(c)}|x_i^{(c)}) = p(n_i = y_i^{(c)} - x_i^{(c)})$.

$$p(y_i^{(c)}|x_i^{(c)}) = \frac{\exp(-\frac{(y_i^{(c)} - x_i^{(c)})^2}{2\sigma_n^2})}{\sqrt{2\pi\sigma_n^2}}.$$  

(3)

Each trellis branch is a transition from one state ($S_j$) at time $i - 1$ to another state ($S_l$) at time $i$. Each branch has an input bit/output codeword bits combination $(u_i, [v_i^{(1)} v_i^{(2)}])$ associated with it, for the rate 1/2 code. A probability metric is found for every branch at every time step, based on

$$\prod_{c=1}^2 p(y_i^{(c)}|x_i^{(c)})$$

(4)

where $x$ is the transmitted BPSK-modulated codeword sequence found from $v$, as $v_i = 1 \rightarrow x_i = 1$ and $v_i = 0 \rightarrow x_i = -1$.

Several choices of metrics are available, including the probability metric of Equation 4, as applied to Equation 2. Other metric choices may be found by taking the log of Equation 4, which gives

$$\sum_{c=1}^2 (y_i^{(c)} - x_i^{(c)})^2$$

(5)

as a metric for each branch, where the overall path metric should be minimized; or the portion of Equation 5 that depends on $x_i^{(c)}$, which is

$$\sum_{c=1}^2 y_i^{(c)} x_i^{(c)}$$

(6)

as a metric for each branch, where the overall path metric should be maximized.

Program Requirements:
1. **Inputs:** Your Viterbi decoding algorithm should take as input a noisy received sequence $y$ and a state transition table that contains the information bit and associated codeword bits for each state transition. You may have generated this state transition table earlier in your encoder design.

2. **Outputs:** Your Viterbi decoding algorithm should output the most likely information sequence $\hat{u}$. It’s OK if your algorithm also outputs the most likely codeword sequence $\hat{v}$ but please note that we don’t actually need to know the codeword $v$; what we are interested in are the original information bits $u$.

**Algorithm:** Your Viterbi decoding algorithm should do the following:

- Start in the zero state $S_0$. You can ensure this while using the complete trellis diagram by setting $\lambda_1(S_j \rightarrow S_l) = -\infty$, $S_j \neq S_0$, if maximizing metrics, or $\lambda_1(S_j \rightarrow S_l) = \infty$, $S_j \neq S_0$, if minimizing metrics. These paths will then be rapidly eliminated from contention once they join other merging paths in $m$ time steps. Note that your representation of "infinity" does not have to be extremely large, merely much larger than any possible contending metric on a merging path, so that the other path will always be chosen.

- Calculate branch metrics $\lambda_i(S_j \rightarrow S_l)$ for each transition from state $S_j$ to $S_l$ at time $i$, $i = 1, \ldots, L + m$. You may choose your branch metric to be of any of the types presented earlier.

- Add branch metrics at time $i$ to previous surviving total path metrics $\Lambda_{i-1}(S_j)$ at time $i - 1$ and ending in state $S_j$ to obtain new total path metrics. If there are two merging paths, one from $S_j$ and one from $S_k$ into state $S_l$, there will be two competing total path metrics for $\Lambda_i(S_l)$: (1): $\lambda(S_j \rightarrow S_l) + \Lambda_{i-1}(S_j)$, and (2): $\lambda(S_k \rightarrow S_l) + \Lambda_{i-1}(S_k)$. These competing path metrics for $\Lambda_i(S_l)$ should be temporarily stored for comparison.

- Compare the two competing total path metrics for $\Lambda_i(S_l)$ merging into the same state $S_l$.

- Choose the largest total path metric (if maximizing the metrics) or the smallest total path metric (if minimizing the metrics), of the competing total path metrics. Eliminate the rest. The surviving total path metric at time $i$ and ending in state $S_l$ becomes $\Lambda_i(S_l)$.

- There are two different methods for obtaining the information bits $\hat{u}$ associated with the surviving path. Implement one of these methods in your program:

  1. **Register-Exchange:** This method uses a sequence of registers at each time step, one for each state (in your program, these registers can be an array/matrix). A register at time step $i$ for state $S_l$ holds an updated list of the information bits from time 1 to the current time $i$ for the surviving path ending in state $S_l$ at time $i$.

This is done as follows:

Start in state $S_0$ with an empty matrix. There are two branches out of state $S_0$ at time 1. If the transition from $S_0 \rightarrow S_0$ is caused by $u_1 = 0$, store 0 in your matrix $u(1, S_0)$. If the transition from $S_0 \rightarrow S_1$ is caused by $u_1 = 1$, store 1 in your matrix $u(1, S_1)$. 
More generally:
Add information bit $u_i$ associated with the surviving branch ending in state $S_i$ to $u(i, S_i)$, so that $u(i, S_i)$ contains all the information bits associated with the path ending in $S_i$ at time $i$. In other words, for a surviving path $S_0 \rightarrow S_1 \rightarrow S_3 \rightarrow S_3$ which ends at $S_3$ in time step 3 and has information bits 1 0 1 associated with these paths, $u(3, S_3) = 101$. The bit 1 was added onto $u(2, S_3)$, because the surviving branch ending in $S_3$ at time step 3 started in $S_3$.

At the end of decoding, the final surviving path will end in $S_0$ if we decode to the end of the codeword (time step $L + m$). The ML estimate $\hat{u}$ will then be contained in $u(L + m, S_0)$.

**Note:** You don’t actually have to store all $L + m$ time steps. However, you want to save at least two time steps, to avoid overwriting information that you need to carry over to the next time step. And in an actual implementation, you would typically use a sliding window decoder (see graduate section), which requires saving a block of $\tau$ time steps.

2. **Trace-Back:** This alternate method does not store the actual information bits, but instead stores information on what the previous state was. This information does not require $2^n$ bits, because the current state as defined by the shift register values AB (for example, $S_1 = 10$) must come from a previous state of either $B_0$ or $B_1$ (in the example, $S_0 = 00$ or $S_2 = 01$). Thus we can define the previous state by only one bit, given the current state.

This method also uses register storage (or an array) to store the bits indicating the previous state. Also as with the register-exchange method, the values stored are cumulative, so that the result at the final time step in $S_0$ holds directions on how to proceed backward through the trellis, along the ML codeword path, to the beginning at $S_0$, time 0.

Note that with this method, after finding the surviving path, the decoder must “trace back” along that path to extract the information bits corresponding to each branch of the path.

There is an advantage to the trace-back method in actual implementation, which is that the contents of the register associated with each state does not have to be shared with another register, but can simply be updated with an additional bit for every decoding time step. The register-exchange method has to transfer the bits in, for example, state 0 to state 1 if the surviving path at time $i$ goes from $S_0 \rightarrow S_1$, before adding the additional bit associated with that branch. This advantage is less obvious when programming the Viterbi algorithm.

- Move to the next time step $i + 1$.
- When you get to the final $m$ time steps, the trellis is limited to paths that move toward state $S_0$ if your codeword is terminated. Again, you may eliminate paths that do not move toward state $S_0$ from contention by setting the branch metrics for those paths to be large and negative (if maximizing metrics) or large and positive (if minimizing metrics).
- At the completion of the final time step $L + m$, you have one surviving path. Choose $\hat{u} = \hat{u}_{L+m}(S_0)$. If you also want to output the most-likely codeword sequence $\hat{v}$, then also choose $\hat{v} = \hat{v}_{L+m}(S_0)$.
- If you have chosen to use the trace-back method instead of register-exchange, then you will now “trace back” the ML path to obtain the information bits associated with each branch of the ML codeword path.

**Exercise:**
1. Write a Viterbi decoding algorithm that meets the requirements for input and output, and performs the above steps as listed for the algorithm.

2. Include your Viterbi decoding program in your report. Please add some comments to your program to enhance readability.

5 Performance Results over an AWGN Channel

We wish to evaluate the performance of the rate 1/2 m=2 convolutional code with generator sequences given in octal as (5₈, 7₈). This code has maximal free distance for its rate and memory, with \(d_{\text{free}} = 5\).

Consider an AWGN channel with noise power \(N_0 = 2\sigma_n^2\), and SNR ranging from 0 to 5 dB, in increments of 1 dB. SNR in dB is defined as \(10\log_{10}(E_b/N_0)\), with \(E_b = E_s/R = 1/R\). Your noise sequence \(n\) is a zero-mean Gaussian, with variance \(\sigma_n^2\).

To simulate the encoding/decoding system, you need to combine your encoder function, AWGN channel function, and Viterbi decoding function. Additionally, you need to generate a uniformly random binary information sequence \(u\) each time to input to the encoder. You may use the random sequence generator from project 1 if you wish.

Exercise:

1. Combine your encoder, AWGN channel emulator and Viterbi decoder into one program. Also, generate a random sequence \(u\) of length 50 for use as input to your encoder.

2. Add a function to your program which calculates both the word error-rate (WER) (count a word error if there is an information bit error in \(\hat{u}\)) and the bit error-rate of the estimated information bits \(\hat{u}\).

3. Include your WER/BER function in your report.

4. Send at least 1000 codewords \(v\) to your decoder. Your WER and BER will be more accurate with more codeword samples.

5. Plot both the codeword error-rate (WER) and the bit error-rate (BER) of the information bits \(\hat{u}\), on the y-axis, vs. SNR on the x-axis. Plot the y-axis on a log scale and the x-axis on a linear scale; this can be done in Matlab with the plotting command \texttt{semilogy()}\). Label both axes, title your plot, and include a legend to distinguish the different curves.

6. To your above plot, add the BER for uncoded BPSK as comparison. Remember that the BER of uncoded BPSK is \(P_b = Q(1/\sigma_n^2)\). (You do not have to simulate the BER of uncoded BPSK; the analytical expression is sufficient). Add this to your plot legend as well.

7. Bonus 1: Calculate the BER and WER for the same nonsystematic rate 1/2 FF encoder, using an information sequence \(u\) bit length of 100 (instead of 50) bits. Do you see any performance improvement in either BER or WER? Why or why not?

8. Bonus 2: Calculate the BER of the systematic rate 1/2 FF encoder with generator sequences given in octal as (4₈, 7₈). Remember that the MSB of the FF binary generator sequence corresponding
to the octal number represents $D^0 = 1$. The systematic rate 1/2 FF encoder has a free distance of $d_{\text{free}} = 4$, which is slightly less than the nonsystematic rate 1/2 FF encoder given earlier. Do you see a difference in performance between the systematic and nonsystematic convolutional codes?

9. **Bonus 3:** Calculate the BER and WER of the same rate 1/2 nonsystematic FF convolutional code with generator sequences given in octal as $(5_8, 7_8)$ if no tail bits are added. Compare this to your earlier results where tail bits are added. Is there any performance improvement? Why or why not? Explain what the effect of adding tail bits to the code is; also explain any disadvantages to tailing that might exist.

10. **Bonus 4:** Compare the BER and WER of the rate 1/2 nonsystematic FF convolutional code to the (8,4) Extended Hamming code used in Project 1, if you ran simulations for the (8,4) EHC over an AWGN channel. Plot BER and WER performance of both codes vs. SNR with axes labeled and a legend as per your other plots. What are the advantages and disadvantages of each code (consider encoding and decoding complexity as well as performance)?

## 6 Graduate Section: Convolutional Codes and Viterbi Decoding: Implementation Issues

Graduate students should do one of the two problems presented in this section. You may do both for extra credit, if you choose, but that is certainly not required.

### 6.1 Distance Spectrum

The distance spectrum of a convolutional code changes with the length of the input information sequence, but each code has a certain minimum Hamming distance $d_{\text{min}}$, or free distance $d_{\text{free}}$ as it is termed for convolutional codes. The entire distance spectrum, even for a predetermined length, is difficult to compute. However, this problem will examine the distance spectrum for only a small number of information bits.

Remember that codeword Hamming distance can be measured with respect to any reference codeword. Thus if we use the all-zeros codeword $C_0$ as our reference codeword, the Hamming distance between $C_0$ and any other codeword $C_j$, $d_H(C_0, C_j)$, can be found simply as the Hamming weight of codeword $C_j$. In other words, if we calculate the weight of the possible codeword sequences corresponding to all possible input sequences of a certain length $L$, we can find the weight distribution which is the distance spectrum of the code (for that fixed length). Obviously if $L$ is very large, this problem becomes intractable.

This problem asks you to calculate the distance spectrum for a short length information sequence, with $L = 4$ and $L = 10$ (plus tailing bits, which do not add to the number of possible information sequences as the tailing bits are completely specified so as to force the codeword to end in the all-zeros state $S_0$).

**Exercise (A) for Distance Spectrum (Weight Enumerator):**

1. Given each of the $2^L$ possible information sequences $u$, find their corresponding codeword $v$ (including tailing bits), and calculate the weight of that codeword. Do NOT print out $u$ or $v$. We are only interested in the weight enumerator $A_d$. The number of times a codeword of weight $d$ appears in the distance spectrum is the weight enumerator $A_d$. 
2. Your program should take as input \( L \), the length of the information sequence \( u \). Your program should output \( A_d \), \( \forall d = 1, \ldots n \cdot L \). For a rate 1/2 code, where rate \( R = k/n \), \( n = 2 \).

3. Run your program for \( L = 4 \) and \( L = 10 \). For each value of \( L \), make a plot of \( A_d \) vs. \( d \) as a bar graph, with \( d \) on the x-axis and \( A_d \) on the y-axis.

4. What is the minimum distance or free distance \( d_{\text{free}} \) of this convolutional code?

Now using your weight enumerators \( A_d \), estimate the probability of a codeword error \( P(E) \) for a terminated codeword sequence with the overbound

\[
P(E) < \sum_{d=d_{\text{free}}}^{n \cdot L} A_d Q\left(\sqrt{\frac{2dRE_b}{N_0}}\right) \tag{7}
\]

This overbound may be simplified by approximating the Q-function \( Q(x) \) as an exponential, where \( Q(x) < \exp(-x^2/2) \). We get the following simpler but slightly weaker overbound:

\[
P(E) < \sum_{d=d_{\text{free}}}^{n \cdot L} A_de^{-\frac{dRE_b}{N_0}} \tag{8}
\]

**Exercise (B) for Word Error Rate Estimation:**

1. Use Equation 7 and the weight enumerators \( A_d \) found previously in Exercise (A) to estimate the WER for the nonsystematic rate 1/2 FF convolutional code with a) \( L = 4 \) and b) \( L = 10 \), with an SNR range from 0 to 5 dB.

2. Use Equation 8 and the weight enumerators \( A_d \) found previously in Exercise (A) to estimate the WER for the nonsystematic rate 1/2 FF convolutional code with a) \( L = 4 \) and b) \( L = 10 \), with an SNR range from 0 to 5 dB.

3. Now plot the WER estimates found above, along with your original WER found in section 5, for SNR from 0 to 5 dB. Use a legend to distinguish the 3 curves. Label all axes and title your plot.

4. How well do the WER estimates work? Is estimate 1 significantly better than estimate 2? Would you consider using estimate 2 to approximate the WER instead of estimate 1? Why or why not?

5. Are the WER estimates better at lower or higher SNR? Explain why.

6. **Bonus:** An improved estimate to Equation 8 can be found as

\[
P(E) < \sum_{d=d_{\text{free}}}^{n \cdot L} A_df\left(d_{\text{free}}RE_b/N_0\right)e^{-\frac{dRE_b}{N_0}}. \tag{9}
\]

where \( f(x) = Q(\sqrt{2x})\exp(-dRE_b/N_0) \).

(a) Use this estimate to calculate the WER, and plot this estimate, along with the 2 previous estimates and your original WER results. Add a legend, label axes, and title your plot.

(b) Is this estimate significantly more accurate than estimate 2?
6.2 Sliding Window Decoder

Sliding window decoding allows a long codeword sequence to be decoded in sections, so that the decoder can output estimates of the early bits without waiting for all bits to come in. Both analytic and experimental results have shown that a window length of $10m$ is sufficient to provide performance very close to no windowing (window length = length of codeword).

There are a few issues involved with windowing. One is that since the codeword has not terminated by the end of the window, the decoder can no longer assume that the ending state is $S_0$, or that the previous $m$ time steps only include paths that lead towards $S_0$. Thus there is no single surviving path ending at $S_0$. How do we then choose the most-likely codeword? The state with the largest metric at the final time step of the window is chosen as the maximum-likelihood path. (This is not the only choice possible, but it is a good one).

It can be shown that with high probability, all of the surviving paths at the end of the window were part of the same original codeword section up until a distance of around $5m$ from the end of the window. Thus the first $5m$ time steps of a $10m$ window should hold reliable information.

Sliding window decoding works as follows:

• A window length of $2\tau$ time steps is chosen.

• Use Viterbi decoding on bits 1-2$\tau$; however, only the first $\tau$ time steps are assumed reliable. Choose the largest metric at time $2\tau$ as the most likely codeword path.

• Save the information bits from only the first $\tau$ time steps of the window.

• Shift the window by $\tau$, to decode bits $\tau + 1$ to $3\tau$. Your starting state at $\tau + 1$ should be the state of the surviving path in the previous window at the end of time step $\tau$ (NOT the end of the previous window!).

• Decode and choose the largest metric at time $3\tau$ as the most-likely codeword path.

• Again, only save the information bits from the first $\tau$ time steps of the new window, that is, from time step $\tau + 1$ to $2\tau$.

• Shift your window by $\tau$, etc.

• If you prefer, you may choose the estimated information bit for time 1, $\hat{u}_1$, from the survivor at time $\tau + 1$, then choose $\hat{u}_2$ at time $\tau + 2$, etc, so that you release a new estimated information bit every time step after $\tau$. Or you may do as in the previous step, which releases $\tau$ estimated information bits, $[\hat{u}_1 \ldots \hat{u}_\tau]$, at time $2\tau$. In this case, the window shifts forward 1 every time step, and a new surviving path is chosen at the end of the window with each time step.

• Continue until you reach the end of the codeword sequence.

Exercise:

1. Write a sliding window function that will take $2\tau$ input bits from the channel, calculate metrics and decode them using the Viterbi decoder, then output estimates for the first $\tau$ information bits. Make $\tau$ an input parameter. Remember that you can no longer assume that the beginning state of the
windowed sequence is $S_0$ or that the ending state is $S_0$. It would be useful to have the beginning state for the window as an input parameter (and thus a useful output parameter might be the state $S_t$ of the surviving path at time $\tau$ of the current window). Include this function in your report.

2. Run your program for three different values of sliding window length: $2\tau = 6$, 10 and 24. Calculate the BER for each sliding window length over a range of SNR from 0 to 5 dB.

3. Plot the BER vs SNR for each window size. Use different line or marker styles to differentiate each curve. Also use a legend to describe which curve is which. Also plot your BER for the unwindowed case in section 5. Do you see much performance loss? What size window would you recommend using for this convolutional code?