EECS 555

Project 1: Simulation of a coded+modulated communication system

Assigned: 1/8, due 2/5

1 Preparatory experiments

1.1 *Generate K realizations of a zero-mean, unit variance Gaussian random variable (In Matlab you can do this using the function* `randn(K,1)`*). From this set of realizations compute an estimate of the $Q(\cdot)$ function* i.e.*,*

$$Q(x) \approx \frac{\textit{number of realizations exceeding } x}{K}$$

*Perform the above experiment for $K = 10^2, 10^3, 10^4, 10^5$, and plot your results on a logarithmic (vertical) axis versus x in dB (*i.e.*, $10\log_{10}(x)$). (In Matlab this can be done using the* `semilogy()` *function). On the same plot show the exact $Q()$ function, which can be obtained using the* `erfc()` *function in Matlab.*

In order to have an accurate estimate of $Q() = 10^{-3}$, approximately how many samples K do we have to consider? How about $Q() = 10^{-4}$?

1.2 *The topic of this problem is a simulation of an antipodal binary transmission scheme. Recall that if BER/SER performance is our objective, then, clearly, we only need to simulate the vector AWGN channel (ie, the pulse shape of the modulation is not of importance as we have seen in 554).*

Some useful suggestions when simulating: *If we simulate using Matlab (this is OK for now but as you will see Matlab is too slow for any useful simulation of practical communication systems for low error rates) then we have to make every effort to "vectorize" our code (i.e., do not simulate the transmission/reception of a single bit N_{total} times, but simulate the transmission of a vector (or packet) of bits of length K for N_{total}/K times). The overall statistical experiment is the same but the latter method is much faster! In general try to eliminate as many for-loops as possible in the critical part of your Matlab code. Also the choice of N_{total} depends on the BER target. Recall from the previous problem that in general you need to collect 10 to 100 errors in order for your simulation results to be accurate. This means that for a BER of 10^{-4} you expect to have 1 error every 10^4 experiments, and thus you need to run about 10^5 to 10^6 experiments. In fact it is much more appropriate that your code counts errors and stops when enough (10 to 100) errors are collected than running a fixed number of iterations. This way the code is adaptive: it spends less time in high BER and more time in low BER.*

Experiment: *Generate K random bits $b_k \in \{0, 1\}, \quad k = 1, \ldots, K$ with equal probability (In Matlab this can be done using several techniques,* e.g.*, you can generate zero mean Gaussian random variables and check their sign, using* `randn(K,1)`*). Collect all these K bits in a vector $\boldsymbol{b}$.*

Based on $\boldsymbol{b}$, generate a vector $\boldsymbol{s}$ where each $s_k \in \{-\sqrt{E_s}, \sqrt{E_s}\}$ (where the signal energy can be normalized such that $E_s = E_b = 1$. Recall that what matters is the E_b/N_0 an not the absolute values of signal and noise).

Generate a vector of K independent zero mean Gaussian random variables $\boldsymbol{n}$, each with variance $\sigma^2 = N_0/2$). The observation equation is $\boldsymbol{z} = \boldsymbol{s} + \boldsymbol{n}$.

At the receiver, standard MAP detection is performed (based on $\boldsymbol{z}$) and an estimate $\hat{\boldsymbol{b}}$ of the transmitted bits is obtained.

The parameter K can be selected according to the memory limitation of the computer, and the experiment should be repeated in order to collect a sufficient number of errors.

By counting the number of disagreements between $\boldsymbol{b}$ and $\hat{\boldsymbol{b}}$, estimate the Bit Error Rate (BER) and plot it in logarithmic vertical axis versus $10\log_{10}(E_b/N_0)$ (i.e., in dB). In the same plot show the exact probability of error (you should get a waterfall curve similar to the ones in our notes/books). (Usually we collect results with some resolution of 0.5 dB on the E_b/N_0 scale and the maximum E_b/N_0 value will be determined by our computing resources, since this results in the lowest BER.)

1.3 *You can now generalize the above simulation program to arbitrary $M-ary$ constellations with arbitrary dimensionality N. You can further generalize so that the simulation program reports both SER and BER results. Do that and obtain BER/SER simulation results for some basic constellations, such as M-PAM (for $M = 4, 8, 16$), M-PSK (for $M = 4, 8, 16$) and M-orthogonal (for $M = 4, 8, 16$). When reporting your results, you should also report any known lower/upper bounds on this performance for comparison.*

This would be a good time to port all your code to C or C++ because from now on the simulations will become more demanding.

2 Simulation of convolutional code with arbitrary modulation and hard/soft Viterbi decoding

2.1 *In this problem we consider a system consisting of an (n, k, K) convolutional code followed by an M-ary modulation scheme (with $M = 2^m$). For simplicity we can assume that n is a multiple of m. Your project is to develop a generic simulation for such a system. Two kinds of receivers will be simulated:*

1. *A separate demodulator providing the Viterbi decoder with hard decisions.*
2. *A combined receiver that performs MAP **decoding** based on the observation from the AWGN channel. This is also called soft-input Viterbi decoding.*

You are supposed to develop a generic Viterbi algorithm for an arbitrary Finite State Machine (FSM). For your convenience I am providing you with a generic FSM class (see fsm.h and fsm.cc files on ctools, as well as an explanation of these in the corresponding help file index.html).

For those of you who didn't take 554, I am attaching a problem that we solved in 554 that has the basic definitions of a FSM and the generic Viterbi decoder description.

EECS 554 Problem:

A generic finite state machine (FSM) is a device with input $x_k \in \mathcal{X} = \{0, 1, \ldots, X-1\}$, state $s_k \in \mathcal{S} = \{0, 1, \ldots, S-1\}$ and output $y_k \in \mathcal{Y} = \{0, 1, \ldots, Y-1\}$, defined by two functions

$$s_{k+1} = ns(s_k, x_k) \qquad \text{(next state function)} \tag{1a}$$

$$y_k = out(s_k, x_k) \qquad \text{(output function)}. \tag{1b}$$

In this formulation, a trellis edge e_k is the pair $e_k = (s_k, x_k)$ that starts at state s_k and ends at state $ns(s_k, x_k)$.

Consider a FSM where an i.i.d. input sequence $\boldsymbol{x} = (x_k)_{k=1}^L$ generates an output sequence $\boldsymbol{y} = (y_k)_{k=1}^L$. The output y_k is transmitted over a discrete input memoryless channel (described by $p(z|y)$). This model is quite general; for instance it models the transmission of a convolutionally encoded sequence, as well as other types of communication systems.

Formulate the Viterbi Algorithm as the solution of the problem of MAP decoding of the sequence $\boldsymbol{x} = (x_k)_{k=1}^L$, i.e., the solution of the problem

$$\hat{\boldsymbol{x}} = \arg\max_{\boldsymbol{x}} p(\boldsymbol{x}|\boldsymbol{z}) = \arg\min_{\boldsymbol{x}} -\log p(\boldsymbol{x}|\boldsymbol{z}).$$

Show that this problem is equivalent to a minimum distance decoding problem. What is the corresponding distance? Why is it additive over time? Using the above formal description of the generic FSM state precisely the VA (in the form of pseudo-code).

Solution:

Let us denote $\boldsymbol{y} \stackrel{\text{def}}{=} cc(s_1, \boldsymbol{x})$ the output sequence produced with input $\boldsymbol{x}$, starting from state s_1. Let us also denote by $I(\cdot)$ the indicator function of its argument. Finally, let $\mathcal{Y}(s_1) \stackrel{\text{def}}{=} \{\boldsymbol{y} | \boldsymbol{y} = cc(s_1, \boldsymbol{x}), \boldsymbol{x} \in \mathcal{X}^L\}$ be the set of all valid output sequences that can be produced with any input sequence $\boldsymbol{x}$ starting from state s_1. Starting from the solution of the MAP problem we have

$$\hat{\boldsymbol{x}} = \arg\max_{\boldsymbol{x}} p(\boldsymbol{x}|\boldsymbol{z}) \tag{2}$$

$$= \arg\min_{\boldsymbol{x}} -\log p(\boldsymbol{x}|\boldsymbol{z}) \tag{3}$$

$$= \arg\min_{\boldsymbol{x}} -\log p(\boldsymbol{z}|\boldsymbol{x}) p(\boldsymbol{x}) / p(\boldsymbol{z}) \tag{4}$$

$$= \arg\min_{\boldsymbol{x}} -\log p(\boldsymbol{x}) \sum_{\boldsymbol{y}} p(\boldsymbol{z}, \boldsymbol{y}|\boldsymbol{x}) \tag{5}$$

$$= \arg\min_{\boldsymbol{x}} -\log p(\boldsymbol{x}) \sum_{\boldsymbol{y}} p(\boldsymbol{z}|\boldsymbol{x}, \boldsymbol{y}) p(\boldsymbol{y}|\boldsymbol{x}) \tag{6}$$

$$= \arg\min_{\boldsymbol{x}} -\log p(\boldsymbol{x}) \sum_{\boldsymbol{y}} p(\boldsymbol{z}|\boldsymbol{y}) I(\boldsymbol{y} = cc(s_1, \boldsymbol{x})) \tag{7}$$

$$= \arg\min_{\boldsymbol{x}} -\log p(\boldsymbol{x}) p(\boldsymbol{z}|cc(s_1, \boldsymbol{x})) \tag{8}$$

$$= \arg\min_{\boldsymbol{y} \in \mathcal{Y}(s_1)} -\log p(\boldsymbol{x}) p(\boldsymbol{z}|\boldsymbol{y}) \tag{9}$$

$$= \arg\min_{\boldsymbol{y} \in \mathcal{Y}(s_1)} \sum_{k=1}^{L} -\log p(x_k) p(z_k|y_k). \tag{10}$$

The last equation shows that the MAP problem can be thought off as a minimization over all valid paths in the trellis of the quantity $\sum_{k=1}^{L} -\log p(x_k)p(z_k|y_k)$. Furthermore, since this quantity is additive in the number of trellis steps (which is a direct consequence of the memoryless channel), this minimization can be performed by the Viterbi algorithm, as long as we identify each trellis transition with the quantity $d(z_k, y_k) = -\log p(x_k)p(z_k|y_k)$. Observe that although $-\log p(x_k)p(z_k|y_k)$ is not a real distance measure between y_k and z_k, it can easily be transformed to one by operations that do not affect the Viterbi algorithm (e.g., adding a constant or multiplying by a positive constant). For instance, in the BSC, and when we have equiprobable inputs, we can add $\log(1-p) + \log(1/2)$ thus getting $-\log p(x_x) - \log p(z_k|y_k) + \log(1-p) + \log(1/2) = -\log p(z_k|y_k) + \log(1-p) = 0$ for $z_k = y_k$ and $-\log p(x_x) - \log p(z_k|y_k) + \log(1-p) + \log(1/2) = -\log p(z_k|y_k) + \log(1-p) = \log \frac{1-p}{p} > 0$ for $z_k \neq y_k$.

We can now state the Viterbi algorithm as follows. We need two storage elements: one array of size S to store the total metric, and one array of size S of lists to store the survivor paths.

1. Initialize:
   $TD(s_1) = 0$.
   $TD(s) = \infty,\ \forall s \in \mathcal{S} \setminus \{s_1\}$.
   $SURVIVOR(s) =$ "$empty\ list$", $\forall s \in \mathcal{S}$.

2. Add-Compare-Select: For $k = 1 \dots L$, and for all $s \in \mathcal{S}$ do:
   $(s^*, x^*) = \arg\min_{(s',x):ns(s',x)=s} TD(s') + d(z_k, out(s', x))$.
   $TD(s) = TD(s^*) + d(z_k, out(s^*, x^*))$.
   $SURVIVOR(s) = (SURVIVOR(s^*), x^*)$.

3. Traceback:
   $s^*_{L+1} = \arg\min_{s\in\mathcal{S}} TD(s)$.
   $\hat{\boldsymbol{x}} = SURVIVOR(s^*_{L+1})$.

In the above we have assumed that the final state is not fixed. The algorithm can be easily modified when the final state is fixed.