ECE300 – Project 2

Brian DeHority, Jonathan Lam, Danny Hong

November 16, 2020

1 Introduction

Digital modulation has become the predominant form of electronic communication, supplanting the ubiquity of non-digital (analog-only) modulation. In contrast to analog modulation, the data sent through digital modulation is sent and received in discretized symbols rather than a continuous stream of information. This project explores several digital modulation systems – binary antipodal pulse amplitude modulation (PAM), binary orthogonal PAM, quadrature amplitude modulation (QAM), phase shift keying (PSK), and differential phase shift keying (DPSK) – and evaluate their relative strengths.

The modulation, transmission (with channel-caused additive white gaussian noise (AWGN)), and demodulation are modeled in MATLAB. Normally, these symbols would be the modulation of a carrier signal with an alphabet of discrete signals, but we only deal with the (discrete) constellation representation of the symbols in the alphabet. These simulations are limited to two-dimensional constellations (defined by the in-phase and quadrature components of the modulated carrier signal). The receiver must use the received, AWGN-distorted signal to decide the most likely transmitted symbol. The decision rule for this project follows a least-squares (LS) decision rule, in which the decision minimizes the (Euclidean) distance error in the constellation space. With higher noise power or closer points on the constellation, the probability of error (intuitively) increases.

2 Review of Digital Modulation Schemes

2.1 Binary Antipodal PAM

The constellation of the binary antipodal scheme are two points equidistant and opposite one another from the origin. (Alternatively, it can be thought of as the special case of PSK, where $M = 2$.) This is a simple binary scheme and maximizes the distance between the two constellation points given the symbol energies. Like binary orthogonal and PSK schemes, the symbols are also equal-energy. The analog representation of this is that the two symbols would be negatives of one another, i.e.:

$$u_m(t) = \pm p(t),\ m \in \{0, 1\}$$

where u_m denotes the analog symbol, and p denotes the unit-energy pulse. The bit error rate (BER) of this scheme is given by:

$$\text{BER} = Q\left(\sqrt{\frac{2\mathcal{E}_s}{N_0}}\right)$$

where $\mathcal{E}_s$ is the energy for symbol (in non-equienergy schemes like generic QAM, this represents the average *average* per symbol) and N_0 is the noise power. We defer the derivation of this formula (and all of the other theoretical calculations of the BER) to the lecture notes or to the textbook.

2.2 Binary Orthogonal PAM

Binary orthogonal has a constellation where the two symbol points are equal energy and at a $\pi/2$ phase shift w.r.t. the origin. This can be thought of as half of a PSK scheme, where $M = 4$. While this is not as robust to noise as binary orthogonal PAM, it is also simple to implement (in-phase (e.g., cos) and quadrature (e.g., sin) components are orthogonal by definition), and it generalizes easily to higher-dimensional spaces. The analog symbols might look like:

$$u_m(t) = p(t)\cos(\omega_c t + \phi_m),\ m \in \{0, 1\},\ \phi_m \in \{\phi_0, \phi_0 + \pi/2\}$$

The BER of this scheme is:

$$\text{BER} = Q\left(\sqrt{\frac{\mathcal{E}_s}{N_0}}\right)$$

Clearly, the difference between this and the binary antipodal scheme is a factor of 2 within the square root in the Q function. This can be seen intuitively with a geometric view: the effective noise power is increased by a factor of $\sqrt{2}$ due to the fact that the distance between the two points (assuming the same symbol energy for both schemes) is reduced by this factor.

2.3 (Phase-Coherent) PSK

PSK constellation points are placed in a circle around the origin. This has two main benefits: all symbols have equal energy, and only the angle (phase) of the received signal is required to make a decision (amplitude is irrelevant in making the decision). In addition, when using greycoding, the most likely error will only be one bit.

$$u_m(t) = p(t)\cos\left(\omega_c t + \frac{2\pi}{M}\right),\ m \in \{0, 1, 2, \ldots, M-1\}$$

To maximize spacing between points (thus minimizing the probability of error), the symbols are equally distributed throughout the circle. The drawback of PSK is that it requires phase coherence; one method to resolve this, as we saw in analog modulation schemes, is the use of a pilot tone. DPSK is one solution to the phase coherence problem (without requiring a pilot tone).

When $M = 2$, the BER for this scheme is the same as that of binary antipodal. When $M > 2$, the BER for this scheme is given by:

$$\text{BER} = 2Q\left(\sqrt{\frac{2\mathcal{E}_s}{N_0}}\sin\frac{\pi}{M}\right)$$

2.4 DPSK

DPSK is similar to PSK, but rather than encoding the information within the angle of the symbol, ϕ_m, it is encoded in the difference in angles between subsequent transmissions ($\Delta\phi = \phi_m - \phi_{m-1}$). While removing the requirement for phase coherence, it transmits one fewer data point for every N symbol transmissions (a negligible loss when N is large). This scheme also roughly doubles the noise power (and thus making it less robust to noise than phase-coherent PSK), since there is variability in the phase of both of the symbols when taking the difference.

For $M = 2$, the BER for this scheme is given by:

$$\text{BER} = \frac{1}{2}\exp-\frac{\mathcal{E}_s}{N_0}$$

For $M > 2$, the BER for this scheme is given by:

$$\text{BER} = 2Q\left(\sqrt{\frac{\mathcal{E}_s}{N_0}}\sin\frac{\pi}{M}\right)$$

Similarly to binary antipodal vs. binary orthogonal schemes, the only difference between this and PSK is the factor of 2 in the sqrt in the Q-function, due to the above reasoning; for every piece of information (i.e., for every differential), there is variability in the phases of two points.

2.5 QAM

QAM encompasses all possible 2-D constellations, and thus all of the previous examples are specific instances of QAM. (However, binary orthogonal signals can easily be extended to higher-dimensional orthogonal constellations, but that is outside the scope of our 2-D constellation project.) In general, in a 2-D constellation space, two basis vectors $\hat{i}$ and $\hat{j}$ are required, and the constellation is of the form:

$$u_m(t) = a(t)\hat{i} + b(t)\hat{j}$$

Of course, since we love our trigonometric orthogonal basis functions, it usually comes in the form:

$$u_m(t) = A_m p(t)\cos(\omega_c t) - B_m p(t)\sin(\omega_c t)$$

The textbook provides equations for the BER when $M = 2^k$, and arranged either in a square (k is even) or a rectangle (k is odd), where the points are all equally-spaced in the quadrature and in-phase components. The benefit of this grid structure is that it requires lower average energy than schemes like PSK, and is simple to produce with the two basis signals.

For square constellations, the BER is (exactly):

$$\text{BER} = 1 - \left(1 - 2\left(1 - \frac{1}{\sqrt{M}}\right) Q\left(\sqrt{\frac{3\mathcal{E}_s}{(M-1)N_0}}\right)\right)^2$$

When k is odd, the textbook provides a "tight upper bound" on the BER:

$$\text{BER} \leq 1 - \left(1 - 2Q\left(\sqrt{\frac{3\mathcal{E}_s}{(M-1)N_0}}\right)\right)^2$$

However, empirically we did not find that this was an upper bound, but rather lower than the empirical values in the $M = 32$ rectangular case. This is discussed further in the Discussion section.

3 Methods

3.1 Least squares decision rule

Our least squares decision rule finds the closest constellation point to a symbol (measured by Euclidean distance) using the `dsearchn` MATLAB function. (The `dsearchn` function uses a Delauney triangulation to efficiently find the nearest point.) This method is faster than using a `for` loop to iteratively find the closest point.

```
% l2_nearest: Finds nearest symbols in a constellation to a set of received
% (noisy) signals
%
% params:
% con   = complex constellation (M x 1)
% est   = complex received signals (N x 1)
%
% returns:
% N_hat = estimated signals
function nearest = l2_nearest(con,  est)

    % transform complex numbers into a 2D real vectors
    con_2d = [real(con) imag(con)];
    est_2d = [real(est) imag(est)];

    % find nearest points (query points are the estimated vectors)
    nearest = con(dsearchn(con_2d, est_2d));
end
```

Figure 1: Least squares decision function

3.2 Transmission simulation

We simulate the transmission of a digital sequence by inputting the base constellation shape, a desired SNR per bit (in decibels), a desired noise power, and the sequence of symbols to transmit (taken from the base constellation). This function does not assume anything about the base constellation; there is no error checking. The base constellation and symbols to transmit are scaled so that the average SNR per bit is the desired value, given by the following equations:

$$\text{SNR per bit} = \frac{E_{avg,sym}}{N_0}$$

$$E_{avg,bit} = \frac{E_{avg,sym}}{\lceil \log_2 M \rceil}$$

The scaled transmitted symbols are now the true transmitted symbols (`true_sym`). Transmission through a noisy channel is approximated by adding 2-D AWGN with variance $\sigma^2 = N_0/2$; since the in-phase and quadrature components are independent additive gaussians, their variances sum;

and since we assume the noise to be isomorphic, each component has half of the variance of the overall AWGN ($\sigma_I^2 = \sigma_Q^2 = \sigma^2/2$). These noisy signals are the simulated received signal; they are then estimated to the nearest constellation points using the least squares decision rule to provide the estimated transmitted vectors (`est_sym`).

There a difference in procedure in simulating DPSK. Firstly, we do not assume phase coherence, so an arbitrary (constant) phase shift is applied to the transmitted vectors. (As shown in the results section, DPSK is not much affected by this, as expected.) This function assumes that the incoming constellation is a correct DPSK constellation (points lie evenly-spaced on a circle centered at the origin, with one of the constellation points lying on the ray $\theta = 0$). The true symbols and transmitted noisy symbols get transformed to their differentials by dividing their each point by the phase of the previous point (division by a complex number causes a phase shift). Only then do the transmitted noisy symbols get estimated to become the estimated transmitted vectors.

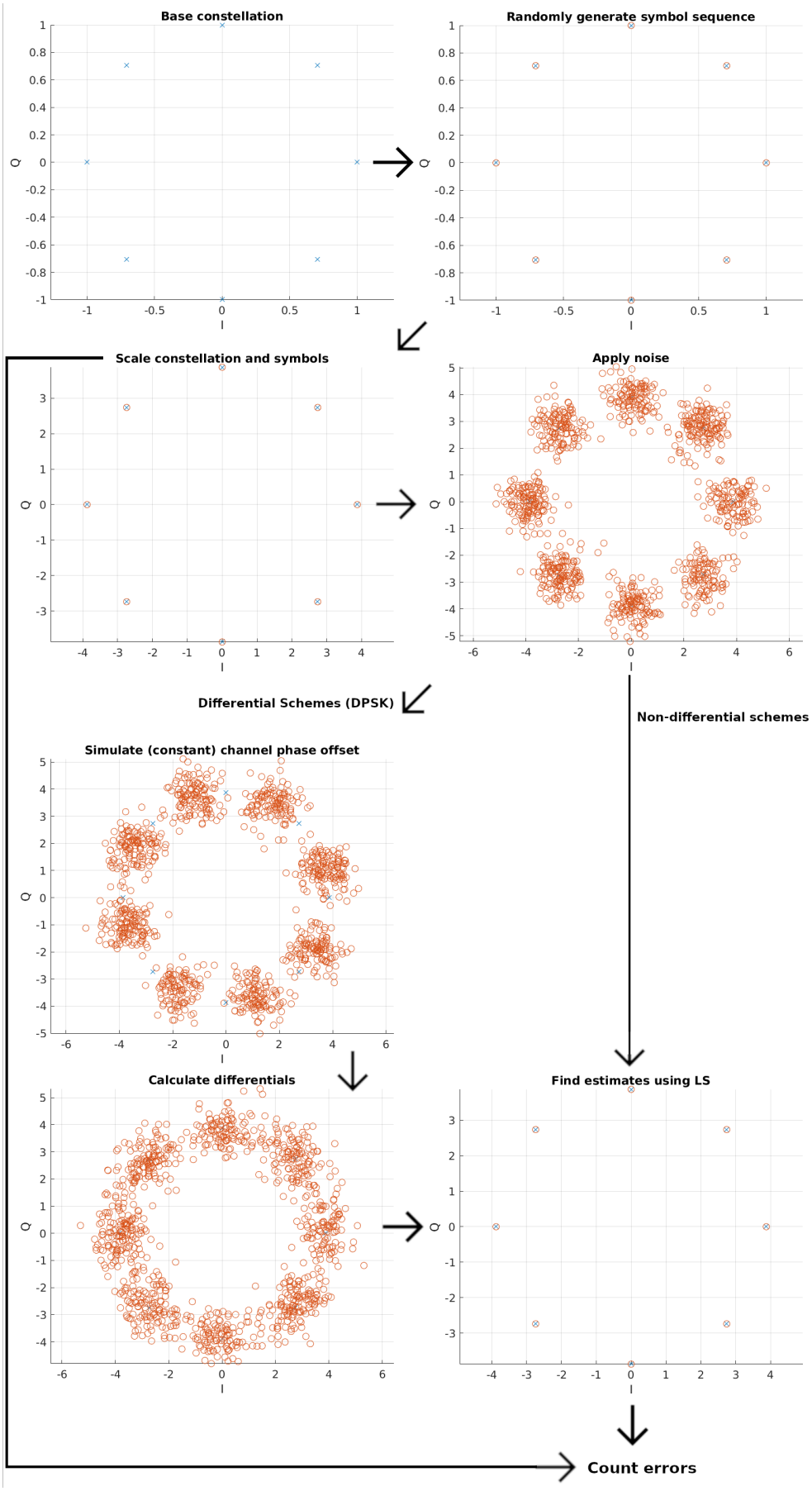

Figure 2: Illustration of the process of simulating and estimating the error for arbitrary constellations. Simulating the differential scheme takes additional work. $M = 8$, SNR=20dB, $N_0 = 1$.

```
% simulate_transmission: simulate transmitting symbols with noise, and
% returns the transmitted symbols and the estimates of the transmitted
% symbols after adding noise
%
% params:
% base_con  = base constellation (M x 1)
% sym       = noise-free transmitted symbols (from base_cons, N x 1)
% N0        = noise power
% SNR       = desired SNR per bit (dB)
%
% returns:
% true_sym  = transmitted (scaled) symbols
% est_sym   = estimated symbols
% scaling_factor    = amount base constellation was scaled by
function [true_sym, est_sym, scaling_factor] ...
    = simulate_transmission(base_con, sym, N0, SNR)

    N = length(sym);
    M = length(base_con);

    % find desired average symbol energy
    E_avg = mean(abs(base_con).^2);
    E_bav = E_avg / ceil(log2(M));

    % using SNR = E_av / N0
    % Ebav = SNR * N0 / log2(M)
    E_bav_des = 10^(SNR/10) * N0;
    scaling_factor = sqrt(E_bav_des/E_bav);

    % scale base constellation and symbols to true constellation
    scaled_con = base_con * scaling_factor;
    true_sym = sym * scaling_factor;

    % produce noise and add to transmitted vectors
    variance = N0;
    noise_proc = sqrt(variance/2) * (randn([N, 1]) + 1j*randn([N, 1]));
    noisy_transmitted = true_sym + noise_proc;

    % determine estimates of transmitted signal
    est_sym = l2_nearest(scaled_con, noisy_transmitted);
end
```

Figure 3: Transmission simulation for phase-coherent (non-differential) schemes

```
% simulate_transmission_diff: simulate transmitting symbols with noise, and
% returns the transmitted symbols and the estimates of the transmitted
% symbols after adding noise; assumes a differential scheme constellation
% (i.e., evenly spaced throughout a circle) where one constellation point
% lies at theta=0
%
% params:
% base_con  = base constellation (M x 1)
% sym       = noise-free transmitted symbols (from base_cons, N x 1)
% N0        = noise power
% SNR       = desired SNR per bit (dB)
%
% returns:
% true_sym  = transmitted (scaled) symbols; first one will be zero for
%             convenience
% est_sym   = estimated symbols
% scaling_factor    = amount base constellation was scaled by
function [true_sym, est_sym, scaling_factor] ...
    = simulate_transmission_diff(base_con, sym, N0, SNR)

    N = length(sym);
    M = length(base_con);

    % find desired average symbol energy
    E_avg = mean(abs(base_con).^2);
    E_bav = E_avg / ceil(log2(M));
    E_bav_des = 10^(SNR/10) * N0;
    scaling_factor = sqrt(E_bav_des/E_bav);

    % scale base constellation and symbols to true constellation
    scaled_con = base_con * scaling_factor;
    true_sym = sym * scaling_factor;

    % produce noise and add to transmitted vectors
    variance = N0;
    noise_proc = sqrt(variance/2) * (randn([N, 1]) + 1j*randn([N, 1]));
    noisy_transmitted = true_sym + noise_proc;

    % randomly rotate symbols to constant channel phase shift (this doesn't
    % really have any real effect, more for show than anything)
    phase_shift = 2*pi*rand();
    noisy_transmitted = noisy_transmitted * exp(1j*phase_shift);

    % translate both noisy_transmitted and true_sym into their
    % phase differentials (their "values"); do this by dividing by the
    % phase component of the previous element
    noisy_transmitted = noisy_transmitted(2:N) ./ ...
        (noisy_transmitted(1:N-1) ./ abs(noisy_transmitted(1:N-1)));
    true_sym = true_sym(2:N) ./ ...
        (true_sym(1:N-1) ./ abs(true_sym(1:N-1)));

    % determine estimates of transmitted signal
    est_sym = l2_nearest(scaled_con, noisy_transmitted);
end
```

Figure 4: Transmission simulation for differential schemes

3.3 Counting number of errors

To determine the number of errors given a sequence of true (`true_sym`) and estimated transmitted vectors (`est_sym`), we use the MATLAB function `nnz`, which finds the number on nonzero elements that exists after taking the difference between the sequence of true symbols and sequence of estimated symbols. A small arbitrary positive threshold of 0.0001 is used to prevent false negatives. This function can be used for both the non-differential or differential schemes, because the estimates of the information (i.e., the differential in the DPSK scheme) was already calculated in the transmission simulation function.

3.4 Generating (base) constellation shapes

The base constellation for binary antipodal and binary orthogonal are known and fixed: $\{-1, 1\}$ and $\{1, j\}$, respectively.

The base constellation for (D)PSK with M symbols is $\left\{\exp j\frac{2\pi}{m}\right\},\ m \in \{0, 1, 2, \ldots, M-1\}$.

For QAM, we used square constellations when M is a square number ($M \in \{16, 64\}$), and a rectangular constellation when M is rectangular ($M = 32$). Our code is optimized only for powers of 2 (square means exponent is even, rectangular means exponent is odd), which is a fair assumption (digital systems, e.g., WiFi, like powers of 2) and lends to the concise constellation-generating code:

```
if mod(log2(M), 2) == 0
    x = (1:sqrt(M)) - (sqrt(M)+1)/2;
    qam_cons = x + x.'*1j;
else
    x = (1:sqrt(M/2)) - (sqrt(M/2)+1)/2;
    y = (1:sqrt(M*2)) - (sqrt(M*2)+1)/2;
    qam_cons = x + y.'*1j;
end
qam_cons = qam_cons(:);     % flatten
```

Figure 5: Generating the QAM constellations (assuming M is a power of 2)

(Additionally, the textbook only gives explicit formulas to estimate the bit error rate when M is a power of 2.)

The base constellations are shown in Figure 6.

4 Results and discussion

The figures on the following pages demonstrate our results: BER as a function of SNR/bit for the different schemes. These compare very closely to the corresponding images in the textbook. The plots are scaled to look like the plots in the textbook, so they do not show the same SNR range. (The textbook is *Communications Systems Engineering, Second Edition.*)

The results mostly match the theoretical when N is large. The confirm the (mostly intuitive) explanations of and the theoretical formulas for the BER provided in the Review of Digital Modulation Schemes section.

Visually, we can see that the binary antipodal and binary orthogonal schemes have a lower BER than the other schemes, but this is mostly because they have so few symbols (M is small). The BER for binary antipodal is lower than that for binary orthogonal, which makes sense because the constellation points are farther from each other. Similarly, the PSK has a lower SNR than the DPSK, which is expected because due to the encoding of information in symbol differentials (but it compromises inherent phase coherence). In general, it is clear that increasing M decreases the robustness to noise.

One error we noticed was that in the QAM $M = 32$ (rectangular) case (see Figure 10), the theoretical equation used was supposed to be a tight upper bound, but it was not always an upper bound. The theoretical estimates are close, but it looks as if the shape is slightly different.

4.1 Code efficiency

The exact techniques were described in the Methods section, and the elapsed time results of running the simulation code on $N = 100000$ symbols is shown in Figure 11. The times shown are not very fast (the visualizations take a few minutes to complete), but they are reasonable considering that running these on domain-specific hardware may allow much better parallelization and other optimizations.

Running the simulations with $N = 10000$ runs (as expected) much quicker and only makes the plots a little noisier. The choice to use $N = 100000$ as opposed to $N = 10000$ for the plots in this report was completely arbitrary.

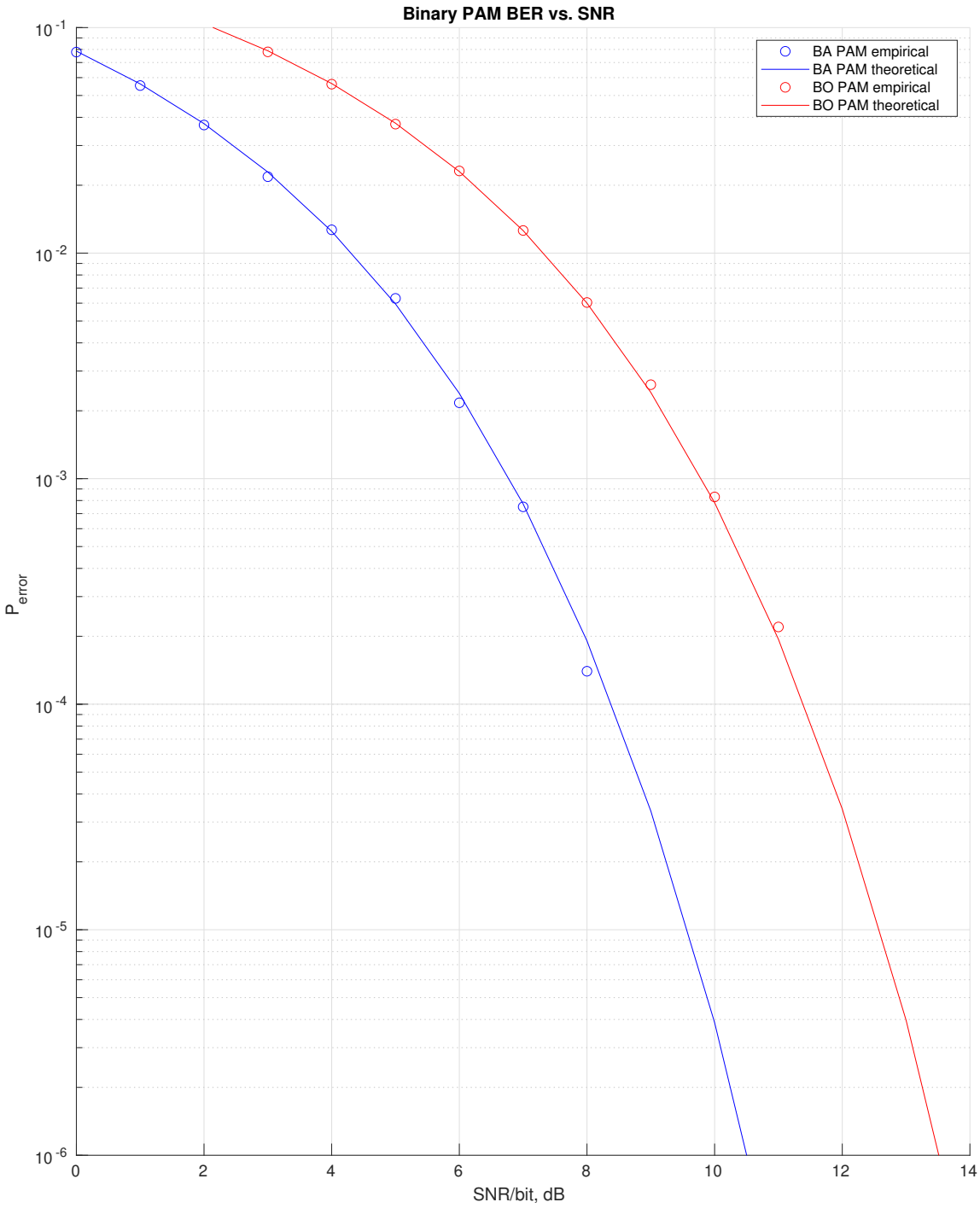

Figure 7: Bit error rate of binary PAM methods. Compare to Figure 7.53 in the textbook.

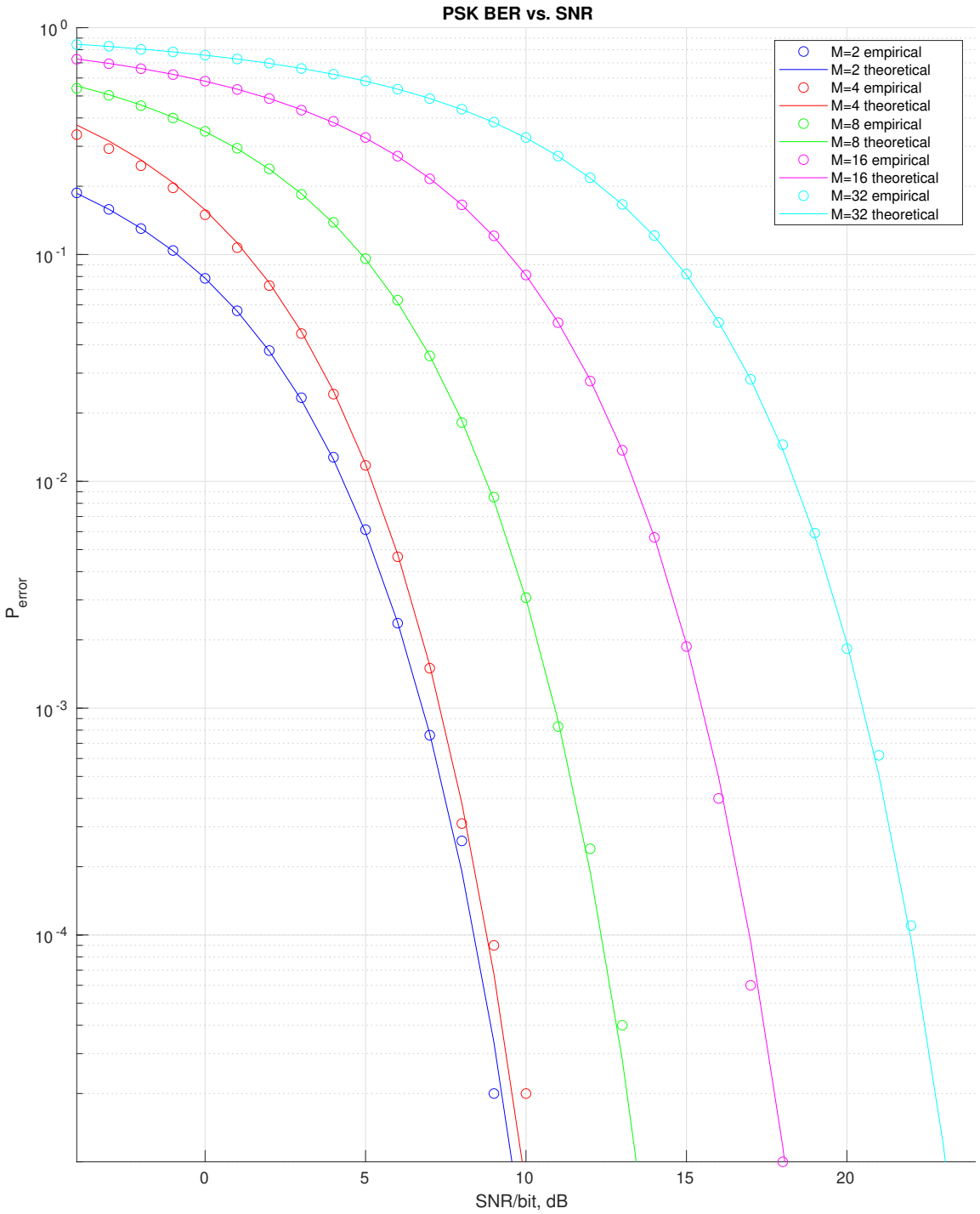

Figure 8: Bit error rate of PSK. Compare to Figure 7.57 in the textbook.

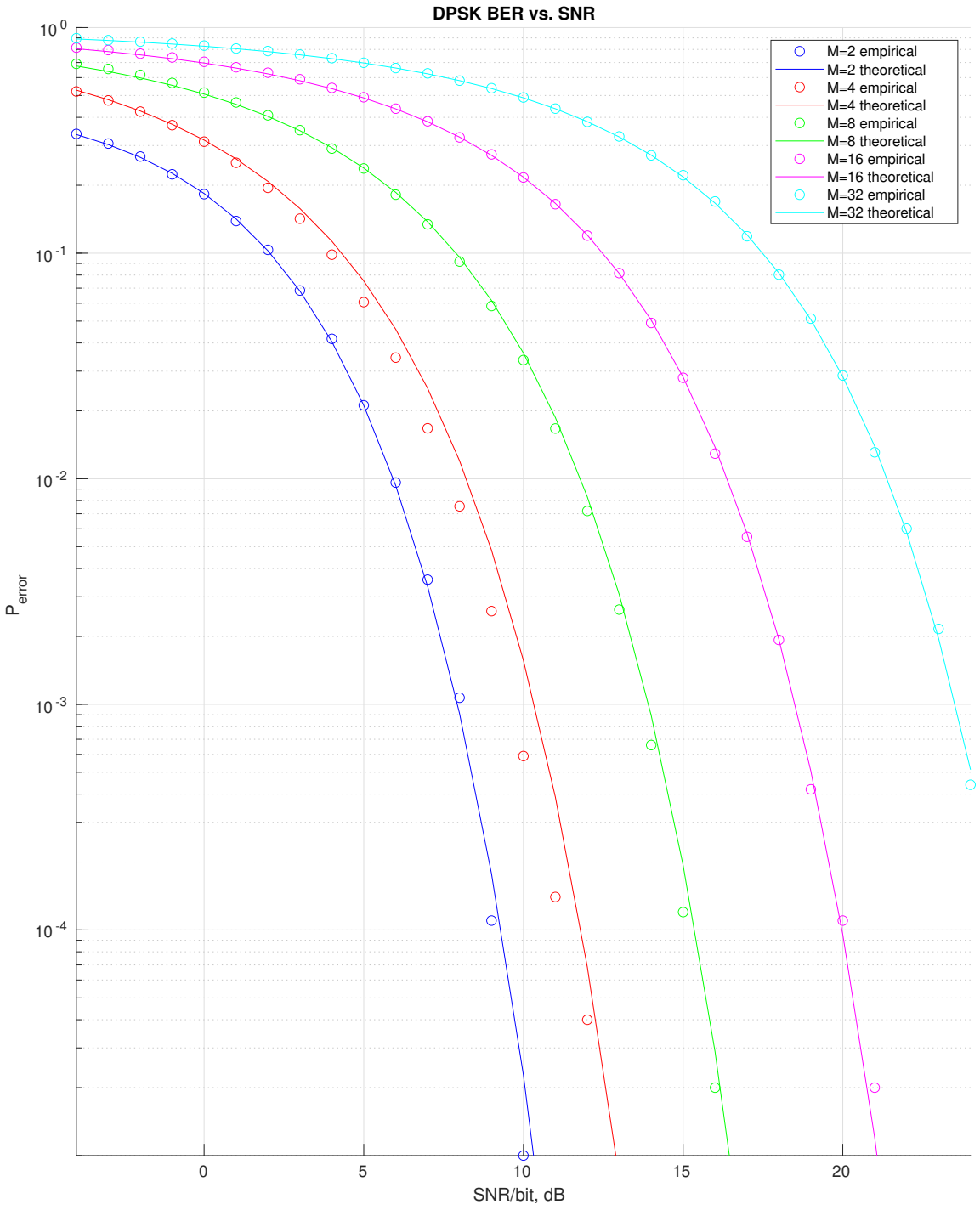

Figure 9: Bit error rate of DPSK. Compare to Figure 7.58 in the textbook.

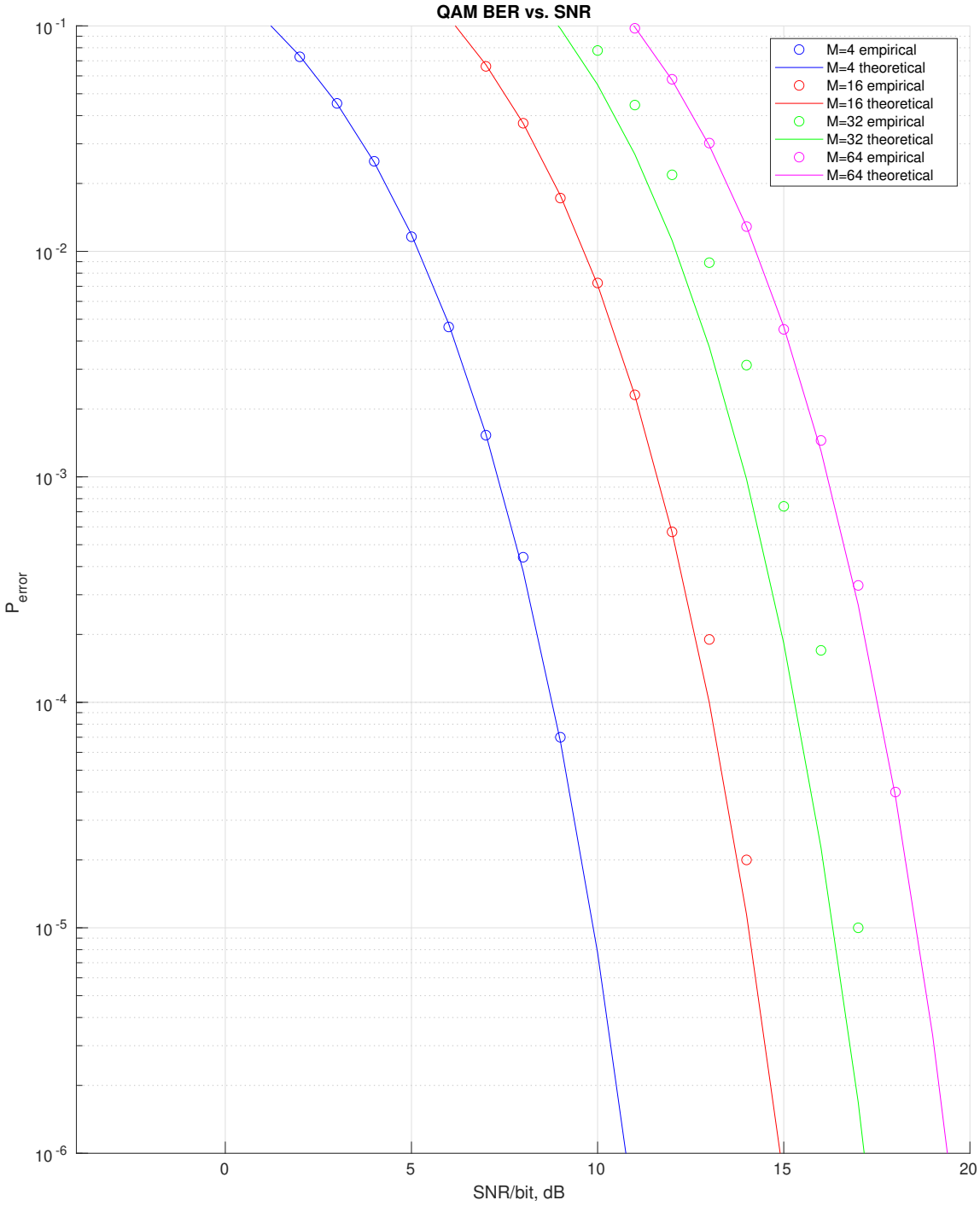

Figure 10: Bit error rate of QAM. Compare to Figure 7.62 in the textbook.

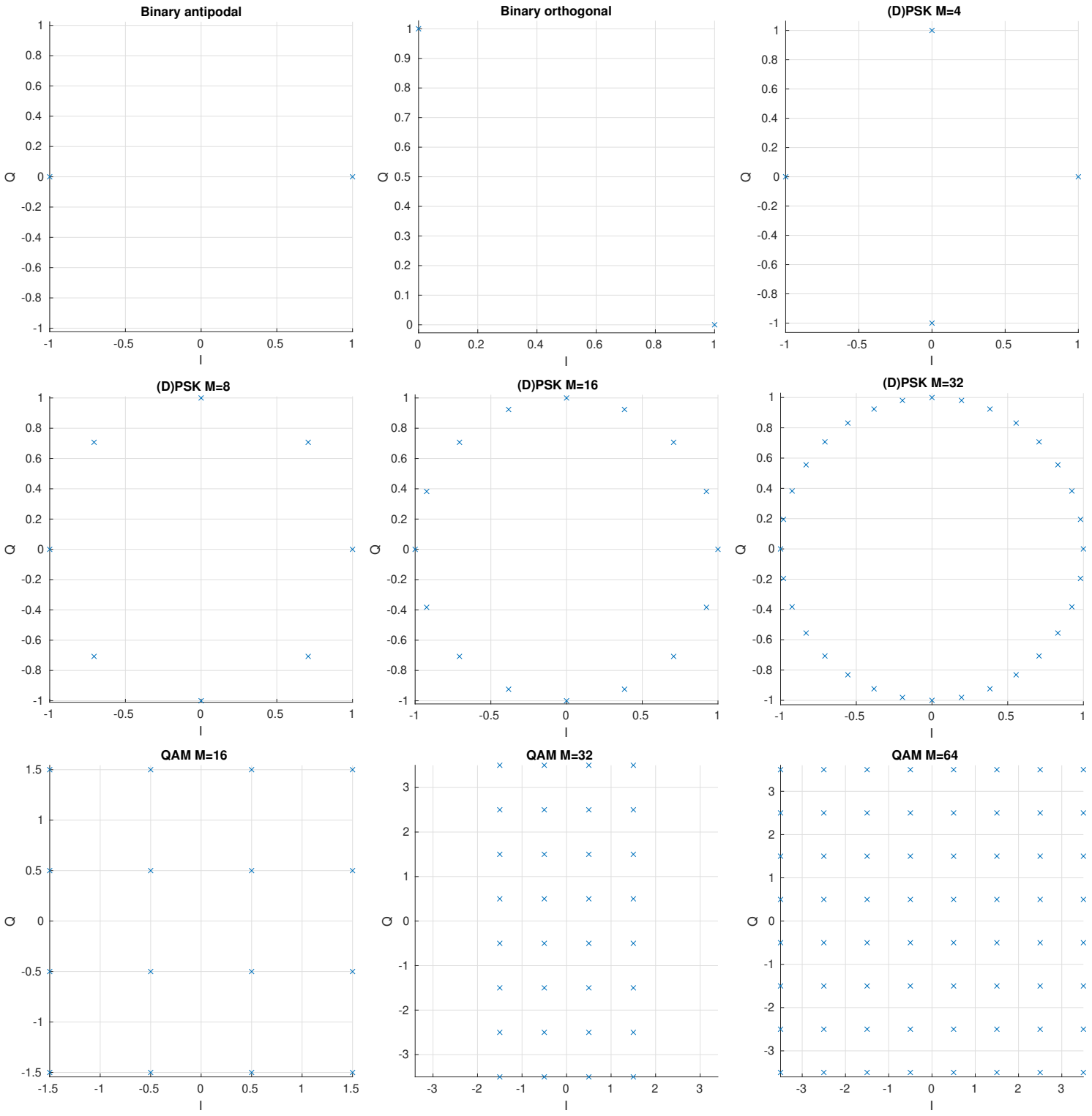

Figure 6: Base constellation shapes

```
Elapsed for binary antipodal for N=100000: 10.33s
Elapsed for binary orthogonal for N=100000: 11.06s
Elapsed for PSK for N=100000: 62.05s
Elapsed for DPSK for N=100000: 62.71s
Elapsed for QAM for N=100000: 44.02s
```

Figure 11: Runtime for each section. Note that (D)PSK generates plots for four different M values, and QAM generates plots for four different (larger) M values.

5 Conclusions and further inquiry

We were able to demonstrate the general relationships between SNR and bit error rate, as well as constellation shape and differential/non-differential coding, using a generic constellation framework. We designed MATLAB procedures that implement a least-squares decision given a generic set of symbols and 2-D constellation, as well as functions to simulate the transmission of given-SNR noisy digital symbols, for both differential and non-differential schemes. The results largely match the theoretical results for bit error rate.

We also did not resolve the error of the estimate for $M = 32$ for PAM (and presumably other rectangular constellations for odd powers of 2). This would be something else to explore in the future.

6 Appendix I: Simulation Code

```
% number of samples
N = 100000;

% noise power
N0 = 1;

tic();

%% binary antipodal

% binary antipodal constellation and random signal sequence
bin_ap_const = [-1; 1];
bin_ap_sym = bin_ap_const(2 * ceil(rand(N, 1)));

% loop through SNRs from -4 to 20
Perrors_ap = zeros(1, 25);
Perrors_theo_ap = zeros(1, 25);
for i = 1:25
    SNR = i-5;

    [true_sym_ap, est_sym_ap, root_Eb] ...
        = simulate_transmission(bin_ap_const, bin_ap_sym, N0, SNR);
    Perror = num_errors(true_sym_ap, est_sym_ap) / N;
    Perrors_ap(i) = Perror;

    Perrors_theo_ap(i) = qfunc(root_Eb*sqrt(2/N0));

end

fprintf('Elapsed for binary antipodal for N=%d: %.2fs\n', N, toc());
tic();

figure('visible', 'off', 'position', [0 0 1000 1200]);
tiledlayout(1, 1, 'TileSpacing', 'Compact');
nexttile();
hold on;
scatter(-4:20, Perrors_ap, 'b');
plot(-4:20, Perrors_theo_ap, 'b');

% binary orthogonal
bi_ortho_cons = [1; 1j];
bi_ortho_m = bi_ortho_cons(2 * ceil(rand(N, 1)));

Perrors_orth = zeros(1,25);
Perrors_theo_orth = zeros(1,25);
for i = 1:25
    SNR = i-5;
    [true_sym_ortho, est_sym_ortho, root_Eb] ...
        = simulate_transmission(bi_ortho_cons, bi_ortho_m, N0, SNR);
    Perror = num_errors(true_sym_ortho, est_sym_ortho)/N;
    Perrors_orth(i) = Perror;

    Perrors_theo_orth(i) = qfunc(root_Eb/sqrt(N0));
end

scatter(-4:20, Perrors_orth, 'r');
```

```
plot(-4:20, Perrors_theo_orth, 'r');
hold off;

title('Binary PAM BER vs. SNR');
ylabel('P_{error}');
xlabel('SNR/bit, dB');
ylim([1e-6 1e-1]);
xlim([0 14]);
set(gca(), 'YScale', 'log');
grid on;    % see note about fudge factor

legend([
    "BA PAM empirical", "BA PAM theoretical", ...
    "BO PAM empirical", "BO PAM theoretical" ...
]);
exportgraphics(gcf(), 'binary.eps');

fprintf('Elapsed for binary orthogonal for N=%d: %.2fs\n', N, toc());
tic();

%% PSK

Ms = [2 4 8 16 32];
colors = ["b", "r", "g", "m", "c"];
figure('visible', 'off', 'position', [0 0 1000 1200]);
tiledlayout(1, 1, 'TileSpacing', 'Compact');
nexttile();
hold on;
for i = 1:length(Ms)
    M = Ms(i);

    % generate basisResults
    theta = linspace(0, 2*pi, M+1);
    theta = theta(1:M).';
    PSK_cons = cos(theta) + 1j*sin(theta);

    % generate signal sequence
    PSK_sym = PSK_cons(ceil(M * rand(N, 1)));

    Perrors_PSK = zeros(1,29);
    Perrors_theo_PSK = zeros(1,29);
    for j = 1:29
        SNR = j-5;
        [true_sym_psk, est_sym_psk, root_Eb] ...
            = simulate_transmission(PSK_cons, PSK_sym, N0, SNR);
        Perror = num_errors(true_sym_psk, est_sym_psk)/N;
        Perrors_PSK(j) = Perror;

        if M == 2
            Perrors_theo_PSK(j) = qfunc(root_Eb*sqrt(2/N0));
        else
            Perrors_theo_PSK(j) = 2*qfunc(root_Eb*sqrt(2/N0)*sin(pi/M));
        end
    end

    scatter(-4:24, Perrors_PSK, colors(i));
    plot(-4:24, Perrors_theo_PSK, colors(i));
end
```

```
hold off;
title('PSK BER vs. SNR');
ylabel('P_{error}');
xlabel('SNR/bit, dB');
set(gca(), 'YScale', 'log');
legend([
    "M=2 empirical", "M=2 theoretical", ...
    "M=4 empirical", "M=4 theoretical", ...
    "M=8 empirical", "M=8 theoretical", ...
    "M=16 empirical", "M=16 theoretical", ...
    "M=32 empirical", "M=32 theoretical"
]);
grid on;
ylim([1e-5 1e-0]);
xlim([-4 24]);
exportgraphics(gcf(), 'psk.eps');

fprintf('Elapsed for PSK for N=%d: %.2fs\n', N, toc());
tic();

%% DPSK

Ms = [2 4 8 16 32];
colors = ["b", "r", "g", "m", "c"];
figure('visible', 'off', 'position', [0 0 1000 1200]);
tiledlayout(1, 1, 'TileSpacing', 'Compact');
nexttile();
hold on;
for i = 1:length(Ms)
    M = Ms(i);

    % generate basis
    theta = linspace(0, 2*pi, M+1);
    theta = theta(1:M).';
    DPSK_cons = cos(theta) + 1j*sin(theta);

    % generate signal sequence
    DPSK_sym = DPSK_cons(ceil(M * rand(N, 1)));

    Perrors_DPSK = zeros(1,29);
    Perrors_theo_DPSK = zeros(1,29);
    for j = 1:29
        SNR = j-5;
        [true_sym_dpsk, est_sym_dpsk, root_Eb] ...
            = simulate_transmission_diff(DPSK_cons, DPSK_sym, N0, SNR);
        Perror = num_errors(true_sym_dpsk, est_sym_dpsk)/N;
        Perrors_DPSK(j) = Perror;

        if M == 2
            Perrors_theo_DPSK(j) = exp(-root_Eb^2/N0)/2;
        else
            Perrors_theo_DPSK(j) = 2*qfunc(root_Eb/sqrt(N0)*sin(pi/M));
        end
    end

    scatter(-4:24, Perrors_DPSK, colors(i));
    plot(-4:24, Perrors_theo_DPSK, colors(i));
```

```
end

hold off;
title('DPSK BER vs. SNR');
ylabel('P_{error}');
xlabel('SNR/bit, dB');
set(gca(), 'YScale', 'log');
legend([
    "M=2 empirical", "M=2 theoretical", ...
    "M=4 empirical", "M=4 theoretical", ...
    "M=8 empirical", "M=8 theoretical", ...
    "M=16 empirical", "M=16 theoretical", ...
    "M=32 empirical", "M=32 theoretical"
]);
grid on;
ylim([1e-5 1e-0]);
xlim([-4 24]);
exportgraphics(gcf(), 'dpsk.eps');

fprintf('Elapsed for DPSK for N=%d: %.2fs\n', N, toc());
tic();

%% QAM
Ms = [4, 16, 32, 64];
colors = ["b", "r", "g", "m"];

figure('visible', 'off', 'position', [0 0 1000 1200]);
tiledlayout(1, 1, 'TileSpacing', 'Compact');
nexttile();
hold on;
for i = 1:length(Ms)
    M = Ms(i);

    % generate constellation (assume M is a power of 2)
    if mod(log2(M), 2) == 0
        x = (1:sqrt(M)) - (sqrt(M)+1)/2;
        % use broadcasting to generate sqrt(M)*sqrt(M) square
        qam_cons = x + x.'*1j;
    else
        x = (1:sqrt(M/2)) - (sqrt(M/2)+1)/2;
        y = (1:sqrt(M*2)) - (sqrt(M*2)+1)/2;
        % use broadcasting to generate sqrt(M/2)*sqrt(M*2) rectangular
        qam_cons = x + y.'*1j;
    end
    qam_cons = qam_cons(:);     % flatten

    % generate signal sequence
    qam_sym = qam_cons(ceil(M * rand(N, 1)));

    Perrors_QAM = zeros(1, 25);
    Perrors_theo_QAM = zeros(1, 25);
    for j = 1:25
        SNR = j-5;
        [qam_true_sym, qam_est_sym, scale_factor] ...
            = simulate_transmission(qam_cons, qam_sym, N0, SNR);
        Perrors_QAM(j) = num_errors(qam_true_sym, qam_est_sym) / N;

        E_avg = mean(abs(qam_true_sym).^2);
```

```
        if mod(log2(M), 2) == 0
            Prootm = 2*(1-1/sqrt(M))*qfunc(sqrt(3*E_avg/((M-1) * N0)));
            Perrors_theo_QAM(j) = 1 - (1 - Prootm)^2;
        else
            Perrors_theo_QAM(j) = 1 - (1 - 2*qfunc(sqrt(3*E_avg/((M-1)*N0))))^2;
        end
    end

    scatter(-4:20, Perrors_QAM, colors(i));
    plot(-4:20, Perrors_theo_QAM, colors(i));
end

hold off;
title(sprintf('QAM BER vs. SNR'));
ylabel('P_{error}');
xlabel('SNR/bit, dB');
set(gca(), 'YScale', 'log');
legend([
    "M=4 empirical", "M=4 theoretical", ...
    "M=16 empirical", "M=16 theoretical", ...
    "M=32 empirical", "M=32 theoretical", ...
    "M=64 empirical", "M=64 theoretical" ...
]);
grid on;
ylim([1e-6 1e-1]);
xlim([-4 20]);
exportgraphics(gcf(), 'qam.eps');

fprintf('Elapsed for QAM for N=%d: %.2fs\n', N, toc());
```