Calvin Passmore

ECE 5660

Homework 4

Problem 1

With reference to the MSK document, write (in good, correct English) re- sponses to the following questions:

(a) Write What is frequency shift keying? What is binary frequency shift keying? (Carefully describe!)

(b) What is the minimum frequency separation ΔF which gives orthogonal signals? Work through the derivation on pp. 2–3 to explain why.

(c) Explain carefully why is it desirable to have continuous phase.

(d) Explain carefully how MSK achieves continuity of phase.

(e) Work through the derivation of the formula θn = θn−1 + an−1π/2 in Eq (3). (show the calculations in the derivation)

(f) Work through the steps to derive Eq (8).

(g) How is MSK both linear and nonlinear modulation?

(a) Frequency Shift Keying is where a change in the carrier frequency indicates a symbol. So the waveform is a carrier with changing frequency, as opposed to changing amplitude such as in PAM.

Binary FSK is where the two bits correspond to a change in frequency delta_F such that representing a 1 means adding delta_F to the nominal frequency F_c and a 0 means subtracting delta_F from F_c.

$s(t) = A \cos[2 \pi (F_c \pm \Delta_F)t]$

(b) The minimum frequency separation = $1 / (4 * T_b) = R_b / 4$. This is because the condition for orthogonality in continuous time is the integral of the two signals must be 0 over the period. Solving the integral gives $\sin(2 \pi (F_i-F_j)T_b) = 0$. $\sin(x) = 0$ when $x = k\pi$ so $2 \pi (F_i-F_j)T_b = k \pi$. We only need one point of the pi circle so choose k=1. Solving the remaining equation gives $F_i-F_j = 1 / (2 * T_b)$.

Then $\Delta_F = (F_1-F_0) / 2 = 1 / (4 * T_b) = R_b / 4$

(c) With continuous phase, the signal avoids sharp changes and the start of one symbol aligns with the next symbol. This reduces the necessary bandwidth to transmit the signal.

(d) MSK achieves continuity of phase by adding or subtracting phase based on the previous phase, and the amplitude of the previous symbol. This way the phase matches the previous phase shifted by its phase shift, equaling them out and creating the continuity.

$\Theta_n = \Theta_{n-1} + a_{n-1}\pi/2$

(e) At any instant $t = (n + 1)T_b$

$2\pi a_n R_b/4\ (t - nT_b) + \Theta_n \mid_{t=(n+1)T_b} = 2\pi a_n R_b/4\ (t - (n+1)T_b) + \Theta_{n+1} \mid_{t=(n+1)T_b}$

$2\pi a_n R_b/4\ ((n + 1)T_b - nT_b) + \Theta_n = 2\pi a_n R_b/4\ ((n + 1)T_b - (n+1)T_b) + \Theta_{n+1}$

$2\pi a_n/4T_b\ (T_b) + \Theta_n = 2\pi a_n R_b/4\ (0) + \Theta_{n+1}$

$\pi a_n/2 + \Theta_n = \Theta_{n+1}$

$\Theta_{n+1} = \Theta_n + a_n\pi/2$

This is non-causal, by reaching into the next symbol. To write it causally, shift n by 1.

$\Theta_n = \Theta_{n-1} + a_{n-1}\pi/2$

(f)

$s(t) = \cos[2\pi F_c t + 2\pi a_n R_b/4\ (t - nT_b) + \Theta_n]$

$= \cos[2\pi F_c t + 2\pi a_n R_b/4\ t - na_n\pi/2 + \theta_n]$ where $na_n\pi/2 + \theta_n = \Theta_n$

$\Theta_n = \theta_n - na_n\pi/2$

$= \theta_{n-1} + a_{n-1}\pi/2 - na_n\pi/2$

$= \theta_{n-1} - a_{n-1}(1-n+n)\pi/2 - na_n\pi/2$

$= \theta_{n-1}-(n-1)a_{n-1}\pi/2+n\pi/2(a_{n-1}-a_n)$

$= \theta_{n-1} + n\ \pi/2\ (a_{n-1}-a_n)$

so when $a_n \neq a_{n-1}$ and when *n* is odd, then $\Theta_n = \Theta_{n-1} \pm \pi$, and Θ_{n-1} otherwise

So now

$s(t) = \cos[2\pi F_c t + 2\pi a_n R_b/4\ (t - nT_b) + \Theta_n]$

$= \cos(2\pi F_c t)\cos(2\pi a_n R_b/4t + \Theta_n) - \sin(2\pi F_c t)\sin(2\pi a_n R_b/4t + \Theta_n)$

$\cos(2\pi a_n R_b/4t + \Theta_n)$

$= \cos(2\pi a_n R_b/4t)\cos\Theta_n - \sin(2\pi a_n R_b/4t)\sin\Theta_n$

$= \cos(2\pi a_n R_b/4t)\cos\Theta_n$

$= d_{I,n}\cos 2\pi R_b t/4$ where $d_{I,n} = \cos\Theta_n$

similarly

$\sin(2\pi a_n R_b/4t + \Theta_n)$

$= \sin(2\pi a_n R_b t/4)\cos\Theta_n + \cos(2\pi a_n R_b t/4)\sin\Theta_n$

$= \sin(2\pi a_n R_b t/4)\cos\Theta_n$

$= \cos\Theta_n \cdot a_n \sin 2\pi R_b t/4$

$= -d_{Q,n}\sin 2\pi R_b t/4$ where $-d_{Q,n} = -a_b\cos\Theta_n = -a_n d_{I,n}$

plugging both d's into s(t) gives

$s(t) = d_{I,n}\cos(2\pi a_n R_b t/4)\cos(2\pi F_c t) + d_{Q,n}\sin(2\pi a_n R_b t/4)\sin(2\pi F_c t)$

using $\sin\alpha = \cos(\alpha - \pi/2)$

$s(t) = d_{I,n}\cos(2\pi a_n R_b t/4)\cos(2\pi F_c t) + d_{Q,n}\cos(2\pi a_n R_b t/4 - \pi/2)\sin(2\pi F_c t)$

$= d_{I,n}\cos(2\pi a_n R_b t/4)\cos(2\pi F_c t) + d_{Q,n}\cos 2\pi a_n R_b t/4(t-T_b)\sin(2\pi F_c t)$

(g) In reality MSK in a nonlinear modulation scheme for a_n. It is only from the viewpoint of d_n that MSK can be seen as linear modulation, seeing it as OQPSK.

Problem 2

With reference to the standards documents: Pick one of the standards and write a brief description (e.g., at least two pages typed) of what the standard is about and briefly describe the major operational blocks in the system.

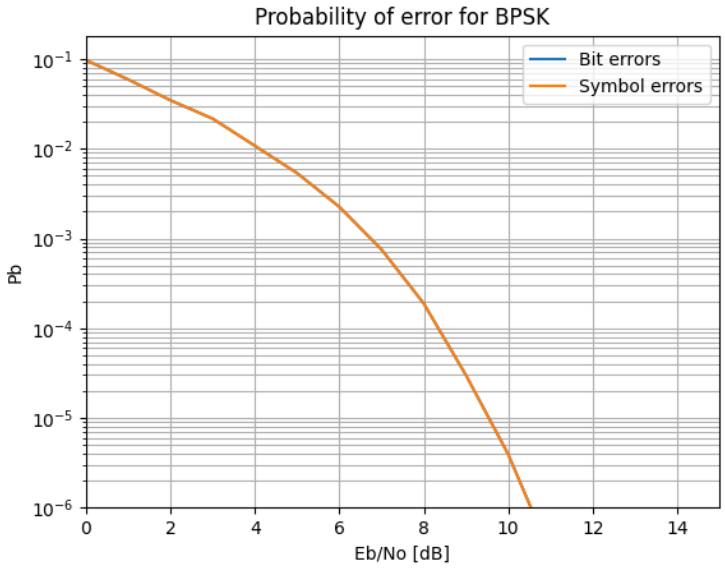
See attached document

Problem 3

(a)

See the attached code. For each constellation type, the LUT, symbol energy, parallel to serial, and slicing had to be done individually. The algorithm to simulate the bits was put into a parent class so that each child class could use the features and data points directly.

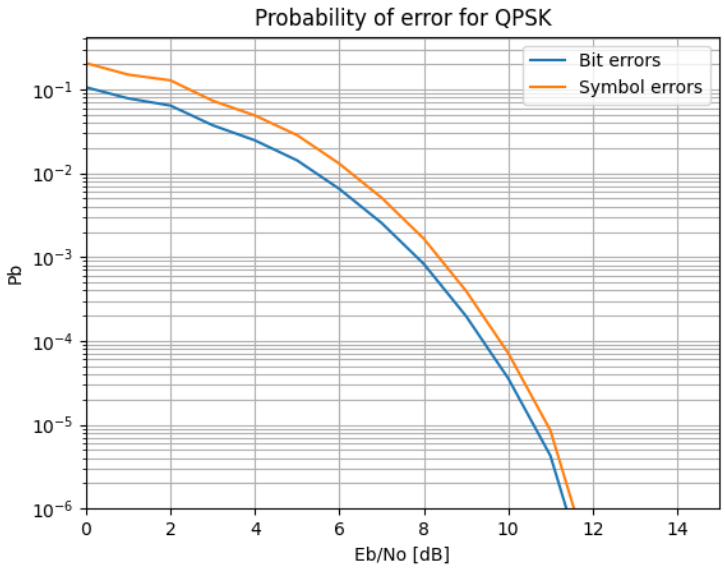
(b)



This is comparable to Figure 6.1.3 in the book. The image in the book seems to be more smooth, indicating they likely used more points to plot it, the points lie in the same general area.

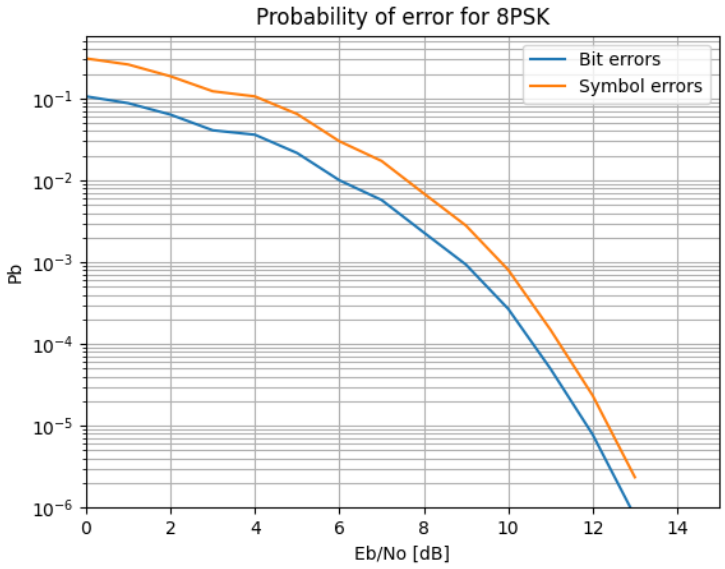
The symbol errors are exactly the same as the bit errors, so only one plot is visible on the graph even though both are plotted. This is due to the one-to-one mapping of symbols to bits in BPSK.

(c)



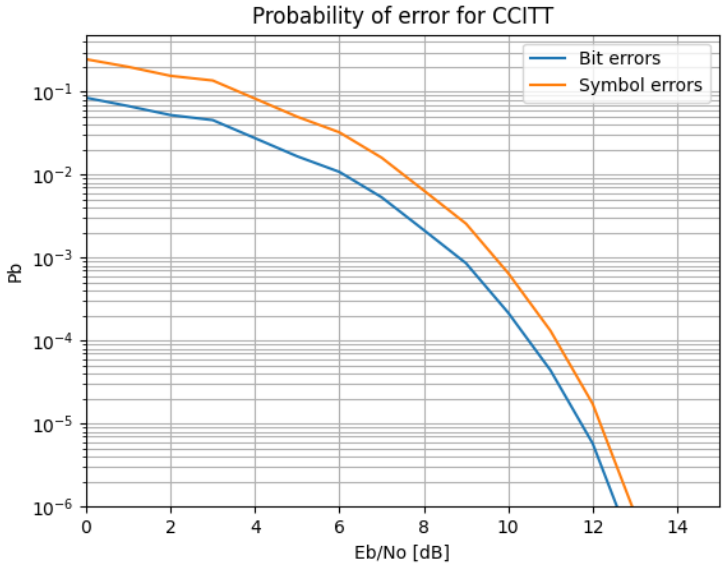
These plots are comparable to the ones in the book, there is a slight variation from what is in the book, and with low Eb/N0 it isn't a very smooth graph.

(d)



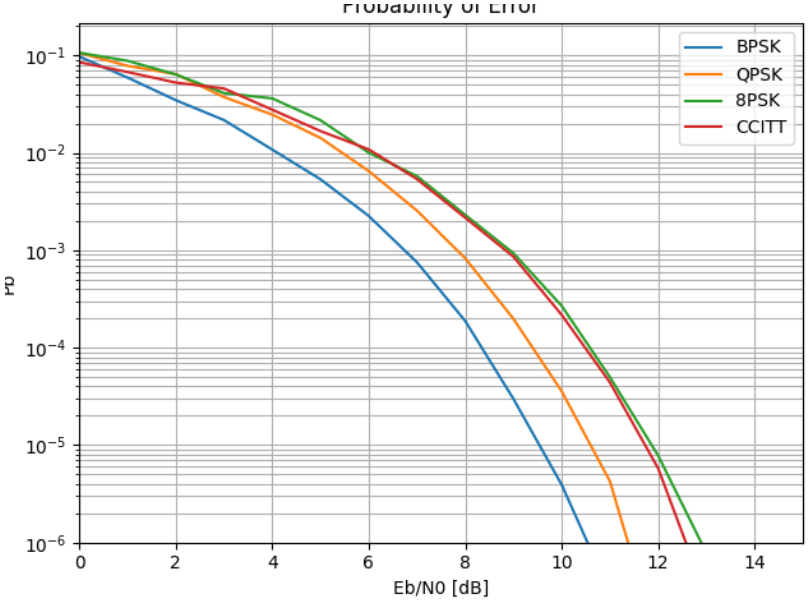
The plots are essentially the same as the ones in the book. There is slight variation due to the randomness of the simulation and the fact the book seems to use more plot points, providing a more smooth and more accurate curve.

(f)



This plot is essentially the same as the book. There is a slight variation due to the randomness of the simulation.

Just for fun, I plotted all of the bit errors together as well. (For some reason the 'savefigure' function in this case cropped it funny)



Code

```
from math import sqrt, cos, sin, floor, ceil
import numpy as np
from numpy.random import rand
import matplotlib.pyplot as plt
import copy

class SignalSpace:
    Es = 0
    Eb = 0
    A = 0
    std_dev = 0
    LUT = []
    bits_per_symbol = 0
    num_axis = 0
    name = "SignalSpace"
    bit_errs = []
    sym_errs = []

    def __init__(self) -> None:
        pass

    def set_symbol_energy(self):
        print("NO GOOD! WASN'T OVERWRITTEN")

    def random_noise(self, std_dev):
        return np.random.normal(loc=0, scale=std_dev)

    def bits_to_amp(self, bits:list):
        """bits is the parallelized version of the bits (as integers)"""
        print("NO GOOD! WASN'T OVERWRITTEN")

    def simulate_single_symbol(self, std_dev, bits):
        """std_dev is the calculated standard deviation and bits is one
```

```
symbol worth of bits, returns number of bit errors and symbol errors"""
        assert len(bits) == self.bits_per_symbol
        sym = self.bits_to_amp(bits)
        sym = [amp + self.random_noise(std_dev) for amp in sym] # Add noise
        bits_hat = self.slice(sym)
        bit_err = 0
        sym_err = 0
        for idx in range(len(bits)):
            if bits_hat[idx] != bits[idx]:
                bit_err += 1
                sym_err = 1 # we want to max out at 1 per call to the
function
        return bit_err, sym_err

    def slice(self, symbol):
        """Turns a single symbol into the closest bits"""
        print("NO GOOD! WASN'T OVERWRITTEN")

    def set_LUT(self):
        """Adjusts the LUT for the given amplitudes"""
        print("NO GOOD! WASN'T OVERRITTEN")

    def plot(self):
        """Plot and save the data"""
        plt.figure()
        plt.clf()
        plt.plot(self.bit_errs)
        plt.plot(self.sym_errs)
        plt.grid(which='both', axis='both')
        plt.yscale('log')
        plt.xlabel("Eb/No [dB]")
        plt.xlim(0, 15)
        plt.ylabel("Pb")
        plt.ylim(1e-6, 0)
        plt.title(f"Probability of error for {self.name}")
        plt.legend(["Bit errors", "Symbol errors"])
        plt.savefig(f"{self.name}.png")

    def simulate(self, dBRangeStart=0, dBRangeEnd=15, Eb=1,
ErrCountTarget=1e2, ErrStopLimit=1e-6):
        self.bit_errs = []
        self.sym_errs = []
        for SNRdB in range(dBRangeStart,dBRangeEnd+1):
            SNR = 10**(SNRdB/10)
            N0 = Eb/SNR
            sigma = sqrt(N0/2)
            # print(sigma)
            bit_err_count = 0
            sym_err_count = 0
            sym_count = 0
            while bit_err_count < ErrCountTarget:
                # print(bit_err_count)
                bits = (rand(self.bits_per_symbol)>0.5).astype(int)
                # print(bits)
```

```
                bit_err, sym_err = self.simulate_single_symbol(sigma, bits)
                bit_err_count += bit_err
                sym_err_count += sym_err
                sym_count += 1
                if bit_err_count > 1 and (bit_err_count/(sym_count *
self.bits_per_symbol)) < ErrStopLimit:
                    break
                print(f"\033[K{self.name:5} SNRdB [{SNRdB}/{dBRangeEnd}]
Error Count [{bit_err_count}/{ErrCountTarget}] {bit_err_count/(sym_count *
self.bits_per_symbol)}", end='\r')
            self.bit_errs.append(bit_err_count/(sym_count *
self.bits_per_symbol))
            self.sym_errs.append(sym_err_count/sym_count)
            # print(f"\n{self.bit_errs[-1]}")

            if self.bit_errs[-1] < ErrStopLimit:
                break
        print("")
        self.plot()
        with open(f"{self.name}_bit.dat", 'w') as w:
            for item in self.bit_errs:
                w.write(f"{item}\n")
        with open(f"{self.name}_sym.dat", 'w') as w:
            for item in self.sym_errs:
                w.write(f"{item}\n")


class BPSK(SignalSpace):
    def __init__(self, Eb=1) -> None:
        super().__init__()
        self.name = "BPSK"
        self.bits_per_symbol = 1
        self.num_axis = 1
        self.Eb = Eb
        self.set_symbol_energy()

    def set_LUT(self):
        self.LUT = [[-self.A], [self.A]]
        # print(f"LUT: {self.LUT} A:{self.A}")

    def set_symbol_energy(self):
        self.Es = self.Eb
        self.A = sqrt(self.Es)
        self.set_LUT()

    def bits_to_amp(self, bits: list):
        assert len(bits) == 1, f"BPSK can only take len 1 bits at a time,
received len {len(bits)}"
        return copy.deepcopy(self.LUT[bits[0]])

    def slice(self, symbol):
        if symbol[0] >= 0:
            return [1]
        else:
```

```
            return [0]

class QPSK(SignalSpace):
    def __init__(self, Eb=1) -> None:
        super().__init__()
        self.bits_per_symbol = 2
        self.name = "QPSK"
        self.num_axis = 2
        self.Eb = Eb
        self.set_symbol_energy()

    def set_LUT(self):
        self.LUT = [[self.A, self.A], [self.A, -self.A], [-self.A, self.A],
[-self.A, -self.A]]

    def set_symbol_energy(self):
        self.Es = self.Eb *1.25
        self.A = self.Es / sqrt(2)
        self.set_LUT()

    def bits_to_amp(self, bits: list):
        assert len(bits) == self.bits_per_symbol
        val = bits[0] << 1 | bits[1]
        return copy.deepcopy(self.LUT[val])

    def slice(self, symbol):
        assert len(symbol) == self.bits_per_symbol
        bits = []
        if symbol[0] >= 0:
            bits.append(0)
        else:
            bits.append(1)

        if symbol[1] >= 0:
            bits.append(0)
        else:
            bits.append(1)

        return bits

class EightPSK(SignalSpace):
    def __init__(self, Eb=1) -> None:
        super().__init__()
        self.bits_per_symbol = 3
        self.name = "8PSK"
        self.num_axis = 2
        self.Eb = Eb
        self.set_symbol_energy()

    def set_LUT(self):
        A = self.A
        for theta in [5*np.pi/4, np.pi, np.pi/2, 3*np.pi/4, 3*np.pi/2,
7*np.pi/4, np.pi/4, 0]:
            self.LUT.append([A * cos(theta), A * sin(theta)])
```

```
    def set_symbol_energy(self):
        self.Es = 3 * self.Eb
        self.A = (-1 + sqrt(1 - 4 * (-6 * self.Eb))) / 2
        self.set_LUT()

    def bits_to_amp(self, bits: list):
        assert len(bits) == self.bits_per_symbol
        val = (bits[0] << 2) | (bits[1] << 1) | (bits[2] << 0)
        return copy.deepcopy(self.LUT[val])

    def slice(self, symbol):
        distances = []
        for look in self.LUT:
            distances.append(sqrt((symbol[0] - look[0])**2) + (symbol[1] -
look[1])**2)
        sym_hat = distances.index(min(distances))
        return [(sym_hat >> 2) & 1, (sym_hat >> 1) & 1, sym_hat & 1]

class CCITT(SignalSpace):
    def __init__(self, Eb=1) -> None:
        super().__init__()
        self.name = "CCITT"
        self.bits_per_symbol = 3
        self.num_axis = 2
        self.Eb = Eb
        self.set_symbol_energy()

    def set_LUT(self):
        A = self.A
        self.LUT = [[2*A, 0], [0, 2*A], [-2*A, 0], [0, -2*A], [A, A], [-A,
A], [-A, -A], [A, -A]]

    def set_symbol_energy(self):
        self.Es = 3 * self.Eb
        self.A = self.Es / (1 + sqrt(2))

    def bits_to_amp(self, bits: list):
        assert len(bits) == self.bits_per_symbol
        val = (bits[0] << 2) | (bits[1] << 1) | (bits[2])
        return copy.deepcopy(self.LUT[val])

    def slice(self, symbol):
        distances = []
        for look in self.LUT:
            distances.append(sqrt((symbol[0] - look[0])**2 + (symbol[1] -
look[1])**2))
        sym_hat = distances.index(min(distances))
        return [(sym_hat >> 2) & 1, (sym_hat >> 1) & 1, sym_hat & 1]

# signal_spaces = [QPSK()]
signal_spaces = [BPSK(), QPSK(), EightPSK(), CCITT()]

for signal_space in signal_spaces:
```

```
    signal_space.simulate(dBRangeStart=0, dBRangeEnd=15,
ErrCountTarget=1e2, ErrStopLimit=1e-6)

plt.figure()
plt.clf()
for space in signal_spaces:
    plt.plot(space.bit_errs)
plt.grid(which='both', axis='both')
plt.yscale('log')
plt.xlabel("Eb/N0 [dB]")
plt.xlim(0,15)
plt.tight_layout()
plt.ylabel("Pb")
plt.ylim(1e-6, 0)
plt.title("Probability of Error")
plt.legend([space.name for space in signal_spaces])
plt.savefig('joined.png')
```

DVB-T300.744

The DVB-T300.744 standard is used for sending and receiving television signals. The standard is designed to be directly compatible with the MPEG-2 format to facilitate easy coding and decoding. The standard consists of the following functional groups: transport multiplexing, outer coding, outer interleaving, inner coding, inner interleaving, mapping and modulation, and Orthogonal Frequency Division Multiplexing (OFDM) transmission. The system allows for different levels of QAM modulation and different inner code rates to help accommodate areas with a higher noise and interference level while allowing for good channels to perform better.

Since this is a broadcast standard, there is no need to support reception at the broadcast location, and the receiving locations don't need to transmit anything. This means the receiver only needs the inverse of the above elements: inner de-interleaver, inner decoder, outer de-interleaver, outer decoder and multiplex adaption. In this summary, however, only the transmit encoding will be covered, and we will assume the input signal/bits are already in the MPEG-2 format.

The outer coding will first apply to the input packet. The Reed-Solomon code will be used with a code length of 204 bytes, and with a dimension of 188 bytes. This means that up to 8 bits can be received incorrectly and it can still be recovered. This also adds some synchronization bytes into the data to help data interpretation and recovery. The outer interleaver will then scramble the bytes.

Interleaving is a process where individual bytes are reordered in a packet upon transport and will be un-reordered upon reception. The purpose of interleaving is to separate burst errors into single

bytes errors that are then easier to correct. The outer interleaver interleaves the Reed-Solomon coded input data into 11 columns. This sequence of bytes is then sent to the inner coder and inner interleaver.

The inner coding uses a range of convolutional coding with rates 1/2, 2/3, 3/4, 5/6, and 7/8. These fractions refer to how many bits are getting input and how many are getting output from the convolutional coding. A code rate of 1/2 means that for every one bit put in, 2 bits are being generated. This is done by setting up a series of flip-flops or other similar delay structures, feeding the bits in then reading out specific delayed bits and Modulo-2 adding them to form the output bits. This can then be undone at the receiver to ensure that all the bits received were the bits sent, potentially correcting any mistakes.

The convolutionally encoded data is then sent to the inner interleaver where multiple layers of interleaving happen. First, there is bitwise interleaving and then symbol level interleaving. The bits are parallelized before bit interleaving, and then given to the symbol interleaver.

The interleaved symbols are then mapped onto the constellation. The available constellations are QPSK, 16-QAM (non-uniform, alpha={2,4}; hierarchical and non-hierarchical, alpha=1), 64-QAM (non-uniform, alpha={2,4}; hierarchical and non-hierarchical, alpha=1). This is then put into the OFDM frame structure, each frame consisting of 68 OFDM symbols.

The OFDM then transmits the data, with a number of sub-carriers dependent on the operation mode, 6817 for 8K mode and 1705 for 2K mode. There are also pilot tones in the OFDM signal

that help with tracking and synchronization. Some of the sub-carriers occasionally won't send data, but instead become a pilot tone for that symbol, which helps synchronization and tracking even more. These boosted-pilot tones reset their pattern in every frame.