BCH

October 12, 2017

1 Simulation von FEC über unterschiedliche Kanäle

1.1 Basic setup

```
In [ ]: import numpy as np
        from pprint import pprint as pp
        %matplotlib notebook
        from matplotlib import pyplot as plt
```

Die Variable `bchlist` enthält eine Liste von Triples von (Paketlänge, Payload, maximal korrigierbare Fehler).

```
In [ ]: # properties of bch as (n, k, t) triples
        bchlist = [ (127, 127, 0),
                (127, 120, 1),
                (127, 113, 2),
                (127, 106, 3),
                (127, 99, 4),
                (127, 92, 5),
                (127, 85, 6),
                (127, 78, 7),
                (127, 71, 9),
                (127, 64, 10),
                (127, 57, 11),
                    ]

        packetlength = 127
        # not elegant: packetlength has to be identical for all BCH codes in the fo

        # let's also try longer packets:
        packetlength = 1023
        bchlist = [
            (1023, 1023, 0),
            (1023, 973, 5),
            (1023, 923, 10),
            (1023, 873, 15),
            (1023, 828, 20),
            (1023, 778, 25),
```

```
        (1023, 728, 30),
        (1023, 698, 35),
        (1023, 648, 41),
        (1023, 608, 45),
        (1023, 573, 50),
    ]


    # how many repetitions to run, for stochastic confidence?
    # also: the more repetition, the more we can find rare events!
    reps = 100000
```

1.2 AWGN channel

Simple iteration over various SNR values

```
In [ ]: snrdB = np.linspace(-5, 15, num=20)
        snr = 10**(snrdB/10.0)

        # bit error rate, assuming BPSK modulation.
        berlist = 0.5 * np.exp(-1*snr)

        # pp(snrdB)
        # pp(snr)
        # pp(berlist)
```

1.2.1 Paket error rate over AWGN

```
In [ ]: def num_errors_awgn(n, ber):
            """For a packet of length n and a given bit error rate,
            randomly generate number of errors in an AWGN channel"""
            return np.random.binomial(n, ber, 1)[0]

In [ ]: def compute_per(bchlist, berlist, reps=reps):
            """For every BCH code in bchlist, and every BER value in
            berlist, simpute reps many packet transmissions.

            A tranmission succeeds if it has fewer simulated errors than
            the considered BCH code can correct.
            """
            per = {}

            for bch in bchlist:
                per[bch] = []
                for ber in berlist:
                    tmp = [num_errors_awgn(bch[0], ber) > bch[2] for i in range(rep
                    per[bch].append( (sum(tmp))/reps)

            return per
```

```
per = compute_per(bchlist, berlist)
```

```
In [ ]: # Let's do a plot: BER as independent variables, for all BCH codes, show
        # packet error curves.
        # Convention: double-logarithmic plots over SNR and PER;
        # we do have logarithmic values for SNR already in the variable SNDdB,
        # so from a plotting tool perspective, this is only a semi-logarithmic plot

        plt.figure()
        for bch in bchlist:
            # Add plotting command here:
        ### YOUR SOLUTION HERE
        plt.title("PER for different FEC schemes (AWGN)")
        plt.xlabel("SNR [dB]")
        plt.ylabel("PER")
        plt.show()
```

1.2.2 Throughput over AWGN

Let us also compute the effective throughput obtained for different BCH codes. On one hand, stronger code reduces error probability; on the other hand, it reduces payload length. So what is optimal code for a given SNR value?

```
In [ ]: throughput = {}
        for bch in bchlist:
            throughput[bch] = []
            for p in per[bch]:
                payload = bch[1]
                packetlength = bch[0]
                # Assign to variable tp the throughput obtained for
                # packet error rate p when there are payload many useful
                # bits in a paket of length packetlength (doh).
                # (It is ok to normalize this to a paketduration here since
                # all our codes have same length.
                # TODO: Think whether this needs extension!)
        ### YOUR SOLUTION HERE
                throughput[bch].append(tp)
```

```
In [ ]: # plot throughput over SNR (logarithmic on SNR, natural unit on throughput)
        plt.figure()
        for bch in bchlist:
            plt.plot(snrdB, throughput[bch])
        plt.title("Throughput relative to uncoded transmission for different FEC sc
        plt.xlabel("SNR [dB]")
        plt.ylabel("Througput")
        plt.show()
```

1.3 FEC over a bursty channel

We switch from a simple AWGN scheme to a Gilbert-Elliot type bursty channel. First, the computation of number of errors in a frame is a bit more complicated. We define a class to hold state information: is the channel in a good or bad state?

```
In [ ]: class GilbertElliot():
            def __init__(self, berGood, berBad, gb, bg):
                self.ber = (berGood, berBad)
                self.M = ( (1-gb, gb), (bg, 1-bg))

                self.state = 0

            def num_errors(self, n):
                """Simulate numer of errors in the next n bits."""
                errors = 0
                count = 0

                while count < n:
                    # how long do we stay in this state?
                    Pnextstate = self.M[self.state][1-self.state]
                    berstate = self.ber[self.state]

                    next_change = np.random.geometric(
                        Pnextstate, 1)[0]

                    # truncate to remaining packet length;
                    # only switch state if this falls indeed inside this packet
                    # Question: Explain why it would not be correct to change state
                    # ALWAYS, irrespective of this test!

                    if next_change <= n-count:
                        self.state = 1-self.state
                    else:
                        next_change = n-count

                    # how many errors within these many bits?
                    next_errors = np.random.binomial(
                        next_change, berstate)

                    errors += next_errors
                    count += next_change

                    # pp((Pnextstate, next_change, next_errors, count, errors))

                return errors

            def steady_state_ber(self):
                # steady states - why is this correct?
```

```
        gb = self.M[0][1]
        bg = self.M[1][0]
        tmp = gb + bg
        Pg = bg / tmp
        Pb = gb / tmp

        # pp((Pg, Pb))
        return Pg*self.ber[0] + Pb*self.ber[1]

    def sim_steady_state_ber(self):
        n = 1000000
        err = self.num_errors(n)
        return err/n
```

Let's get an example channel object and look at its long-term steady-state, average bit error rate.

```
In [ ]: ge = GilbertElliot(berGood=10**-6, berBad=2*10**-1,
                           gb = 1- 0.999, bg = 1-0.99)
        ss_ber = ge.steady_state_ber()
        print(ss_ber)
        sim_ber = ge.sim_steady_state_ber()
        pp(sim_ber)
```

So it turns out that these parameters give us a pretty bad channel, but to illustrate the effects, that's ok)

1.3.1 A histogram for a GE channel

What is the distriubution of number of errors in a packet under such a channel assumption?

```
In [ ]: errors = [ge.num_errors(packetlength) for i in range(100000)]
        ## pp(errors)
        plt.figure()
        n, bins, patches= plt.hist(errors, bins=range(max(errors)))
        plt.show()
```

```
In [ ]: pp(n)
        pp(bins)
        pp(sum(n))
        testber = sum([x*y for x, y in zip(n, bins)])/sum(n)/packetlength
        pp(testber)
```

Compare against an AWGN channel with the same BER:

```
In [ ]: ge_ber = ge.steady_state_ber()
        errors = [num_errors_awgn(packetlength, ge_ber) for i in range(10000)]
        # pp(errors)
        plt.figure()
        n, bins, patches= plt.hist(errors, bins=range(max(errors)))
        plt.show()
```

```
In [ ]: pp(n)
        pp(bins)
        testber = sum([x*y for x, y in zip(n, bins)])/sum(n)/packetlength
        pp(testber)
```

What is the interpretation here?

1.3.2 YOUR SOLUTION HERE

1.3.3 Use bursty channel with FEC codes

```
In [ ]: def compute_bursty_per(bchlist, ge):
            per_bursty = []
            per_awgn = []

            ss_ber = ge.steady_state_ber()

            for bch in bchlist:
                tmp = [ge.num_errors(bch[0]) > bch[2] for i in range(reps)]
                per_bursty.append(sum(tmp)/reps)

                tmp = [num_errors_awgn(bch[0], ss_ber) > bch[2] for i in range(reps
                per_awgn.append(sum(tmp)/reps)


            return (per_bursty, per_awgn)


        per2 = compute_bursty_per(bchlist, ge)
        pp(per2)
```

```
In [ ]: # to plot, let's first get a good horizonatel axis:
        t = [bch[2] for bch in bchlist]
        plt.figure()
        plt.semilogy(t, per2[1], label="AWGN")
        plt.semilogy(t, per2[0], label="bursty")
        plt.ylabel("PER after FEC")
        plt.title("PER for BCH AWGN and bursty channel")
        plt.xlabel("Number of correctable bits")
        plt.legend()
        plt.show()
```

```
In [ ]: # plot throughput
        throughput_awgn = [ (1-per)*(bch[1]/bch[0])
                           for bch, per in zip(bchlist, per2[1])]
        throughput_bursty = [ (1-per)*(bch[1]/bch[0])
                             for bch, per in zip(bchlist, per2[0])]

        plt.figure()
```

```
plt.plot(t, throughput_awgn, label="AWGN")
plt.plot(t, throughput_bursty, label="bursty")
plt.title("Throughput for BCH AWGN and bursty channel")
plt.ylabel("Throughput after FEC")
plt.xlabel("Number of correctable bits")
plt.legend()
plt.show()
```

2 Delay

And what about delay characteristics? We save this for anoth