AIMS DMG Sage Demonstration 17
AIMS 2013-14: Designs, Matroids and Graphs

Rob Beezer
University of Puget Sound

Nancy Neudauer
Pacific University

January 17, 2014

1 A Design that Yields a Good Nonlinear Code

We start with a design and finish with a code. More specifically, the (unique) $2 - (11, 5, 2)$ design. We build it with *nonzero* quadratic residues mod 11 (from a finite field).

```
F = GF(11)
quad_res = Set([x^2 for x in F if x != 0])
quad_res
```

Use as a difference set.

```
blocks = [[x + y for y in quad_res] for x in F]
blocks
```

This is a 2-design, with a transitive automorphism group.

```
B = BlockDesign(11, blocks)
B.is_block_design()
```

```
G = B.automorphism_group()
G
```

```
G.is_transitive()
```

We use the blocks to indicate the locations of 1's in a codeword, and we add one special codeword, the all 1's codeword. We also drop the first coordinate of these vectors, so the codewords have length 10, not 11.

Because the automorphism group is transitive, deleting any other coordinate would have the same effect. Our computations will be easier if we force our codewords into a vector space over $GF(2)$, so we set that up now.

```
V = GF(2)^10
```

```
# Add list with all elements
# Will create all 1's codeword later
blocks = blocks + [F.list()]
```

```
blocks
```

```
code = []
for b in blocks:
    codeword = [0]*11
    for i in b:
        codeword[i] = 1
    # drop coordinate 0
    # binary vectors
    code.append(V(codeword[1:]))
print Sequence(code, cr=True)
```

The code should have a “cyclic” feel to it, as each codeword is very close to a shift of the previous one (except for the all 1’s codeword).

As this is not a linear code (it is not the kernel of a matrix, nor a subspace of $GF(2)^{10}$) to get the minimum distance we must compare *all* pairs of different codewords. If we do not write this as a list comprehension, it takes six lines of code.

```
distances = [len((u-v).nonzero_positions()) for u in code for v in code if u != v]
d
```

We can scan through the pairwise distances to look for the smallest.

```
distances
```

Or have Sage do it for us.

```
d = min(distances)
```

Boom!

So we have a code with minimum distance $d = 5 = 2(2) + 1 = 2e + 1$, which can therefore correct up to 2 simultaneous errors in a codeword of length 10.

Suppose any given bit has a 99% chance of being corrupted. We can correct 0 errors, 1 error, 2 errors and have an error-free transmission. How often does this happen?

```
p = 0.99
p^10 + binomial(10,1)*p^9*(1-p)^1 + binomial(10,2)*p^8*(1-p)^2
```

So 2 times in $10,000$ we have more than two errors, and we cannot correct the mistake. Is this good enough? Depends. Are you transmitting vacation photographs or financial transactions?

We can detect up to 4 errors, however, and perhaps ask for a retransmission. The probability of 4 or fewer errors is:

```
p = 0.99
sum([binomial(10,i)*p^(10-i)*(1-p)^i for i in range(0,d)])
```

25 in a billion times, errors will go undetected.

What price do we pay for this redundancy? We are using 10 bits of bandwidth to send one symbol out of 12 (we have just 12 codewords). Two ways to view this — we could send $2^{10} = 1024$ symbols with 10 bits, or our 12 codewords require at most 4 bits since $2^4 = 16$. So we are using more than 2.5 times as much capacity as we would need if our communication medium was perfect. The right factor to compute is:

```
N(10/log(12, 2))
```

One billion transmissions, $10/\log_2(12)$ bits of content per transmission, 8 bits per byte, $\left(2^{10}\right)^3 = 2^{30}$ bytes in a gigabyte, so we get 25 errors in the following number of gigabytes of *content*.

```
((10^(9)*10)/(8*2^(30))/N(10/log(12, 2))
```