Week 3: Data errors

Digital data comes in a variety of forms:

– Numerical data
- eg. Numbers collected from some sampling device
- Easily read by programs
- Data errors generally related to instrument uncertainties

– Categorical data
- eg Classifications, Groupings, etc
- Easily read and categorised by programs
- Wrong-category Error is more problematic (how to correct?)

– Descriptive / formatted / structured data
- eg free text, XML, JSON, special file formats
- Requires syntax (structure) & semantic (meaning) checking

Binary data errors

- All the data errors in the previous slide rely on the bits themselves being correctly transmitted.
- What if they are not?
  – Firstly can we tolerate bit errors?
    - eg Video streams have built in error tolerance and recovery
  – Secondly can we detect bit errors?
    - This may be sufficient, as we can ask to retransmit the data
  – Thirdly, can we correct bit errors?
    - If retransmission is not possible or practical (eg Big Data), then can we at least correct some of it?

Data error tolerance

- Most binary data is structural
  – Data structures include field lengths, data sizes, formatting rules
  – Having bit errors in structured streaming usually means the downstream system gets confused as to where it is up to and so may need to resynchronise
  – So error tolerance can mean
    - wrong pixel values in images video (only visible to data consumers, not actually a system error.
    - discovery that the system is confused and needs to recover
  – In the case of MPEG video, all stream data headers start with 12 1 bits in a row, which never happens in the data itself.
    - so if a video client gets confused, it simply waits for the next set of 12 1 bits, and restarts from there.
- But for purely numerical data, we can use Gray codes.

Data error tolerance: gray code

- When numeric data is digitally sampled from an instrument, you cannot be sure if any error is due to but errors in transmission. But you can limit its effects.
- One way we can limit the effect of such errors is for the device to use Gray codes instead of binary.
- Consider the following:

  SourceBin → 0111 → 1-bit error → 1111 → Dest
  SourceGray → 0100 → 1-bit error → 1100 → Dest

  – Clearly in the first case the 1 bit error has caused the reading to be almost double the correct value, from 7 to 16.

  – In the second case, the same value (7) as Gray code suffer the same bit error, but the new value is 8 in Gray Code.

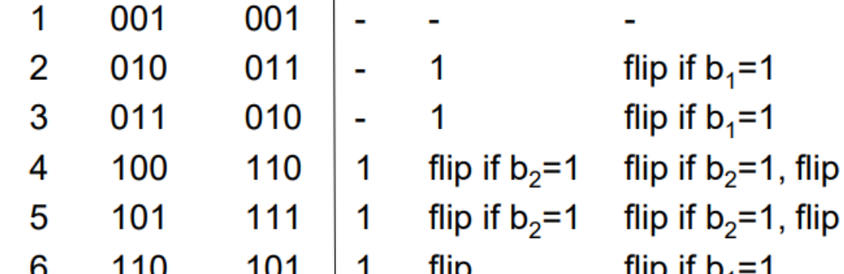
- Gray codes are designed so that each value differs from its neighbour by only a single bit., so that single bit errors will only change the value by +/- 1.
- Consider the following truth table (b_2-b_0 are bits of Val)

Val	Bin	Gray	b_2	b_1	b_0
0	000	000	-	-	-
1	001	001	-	-	-
2	010	011	-	1	flip if b_1=1
3	011	010	-	1	flip if b_1=1
4	100	110	1	flip if b_2=1	flip if b_2=1, flip if b_1=1
5	101	111	1	flip if b_2=1	flip if b_2=1, flip if b_1=1
6	110	101	1	flip	flip if b_1=1
7	111	100	1	flip	flip if b_1=1

- Now extend the pattern to b3 of Val.
  – Does the same pattern hold?

Val	Bin	Gray	Val	Bin	Gray
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

Shown below are the logic gates for conversion to/from Gray codes, and equivalent code.




```
int function Binary_To_Gray(int num) {
    return num^ (num >> 1);
}
int function Gray_To_Binary(int num) {
    int prop = num >> 1;      // Simulate right propagation
    while prop != 0
        num = num ^ prop;
        prop = prop >> 1;
    }
    return num;
}
```

- Some interesting properties of Gray codes
  – The Numerical difference between adjacent values is 1
  – The NHamming distance (see later) between adjacent values is 1
  – A permutation of the columns of a Gray code is still a Gray code:

0,1,2,3 → 00,01,11,10,
swap columns,
00, 10,11,01 → 0,2,3,1

– You could use Gray codes for bit-error tolerant encryption

Data error detection: parity

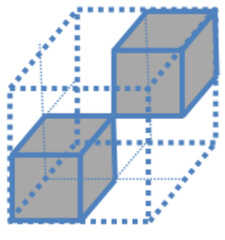
- A simple way to handle bit errors is to count the bits and force the correct number to always be even (or odd), by adding a parity bit if needed to force this.
- Redundant bits can be added to data, to detect errors
  – e.g. adding a parity bit to 7-bit ASCII
  – parity bit is MSB
- ASCII with even parity
  – total number of 1s in a (8 bit) character is even
    - 'A' = 100 0001, parity 0, code = 0100 0001
    - 'C' = 100 0011, parity 1, code = 1100 0011
- ASCII with odd parity
  – parity bit is set so total number of 1s is odd

Parity

- Any 1 bit error will change the number of 1s and 0s, so the parity bit will no longer be correct
  – error is thus detected
  – but not enough information to correct error (which bit has been flipped?)
- Flipping a single bit of an ASCII character produces another valid ASCII character
  – there is a 1-bit 'distance' between valid ASCII symbols (characters)
- If we could increase the 'distance' (in bits) between valid symbols, then a single bit error would turn a valid symbol into an invalid symbol

Data error correction: hamming distance

- The 'bit distance' between valid symbols in a code is called the Hamming distance
- e.g. say a code only had the symbols 000 and 111
  – everything else is illegal
  – so the Hamming distance is 3 (need 3 flips to move from one valid symbol to another)
- If 1 bit is flipped, and we receive 001:
  – we know an error has occurred
  – we know the correct code must have been 000
  – such a code allows error *correction*
- This is generalisation of Gray codes



* David Hamming was a Mathematician during WW2