
Constrained Coding Techniques for Advanced Data Storage Devices

Paul H. Siegel

Director, CMRR

Electrical and Computer Engineering

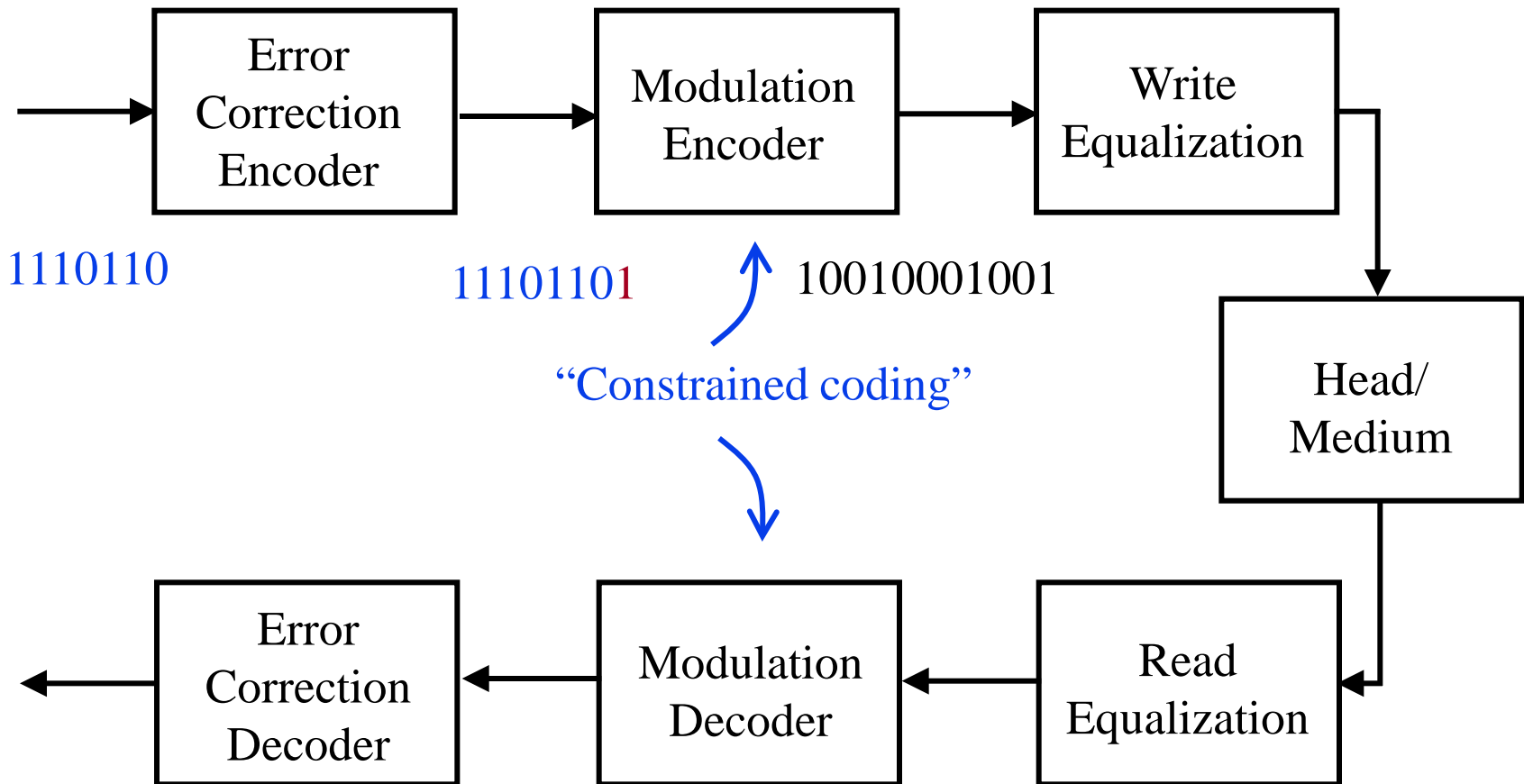
University of California, San Diego



Outline

- Digital recording channel model
- Constrained codes (track-oriented)
 - Bit-stuffing, bit-flipping, symbol-sliding
 - Information theory of constrained codes
- Constrained codes (page-oriented)
 - Bit-stuffing in 2-dimensions
- Concluding remarks

Digital Recording Channel

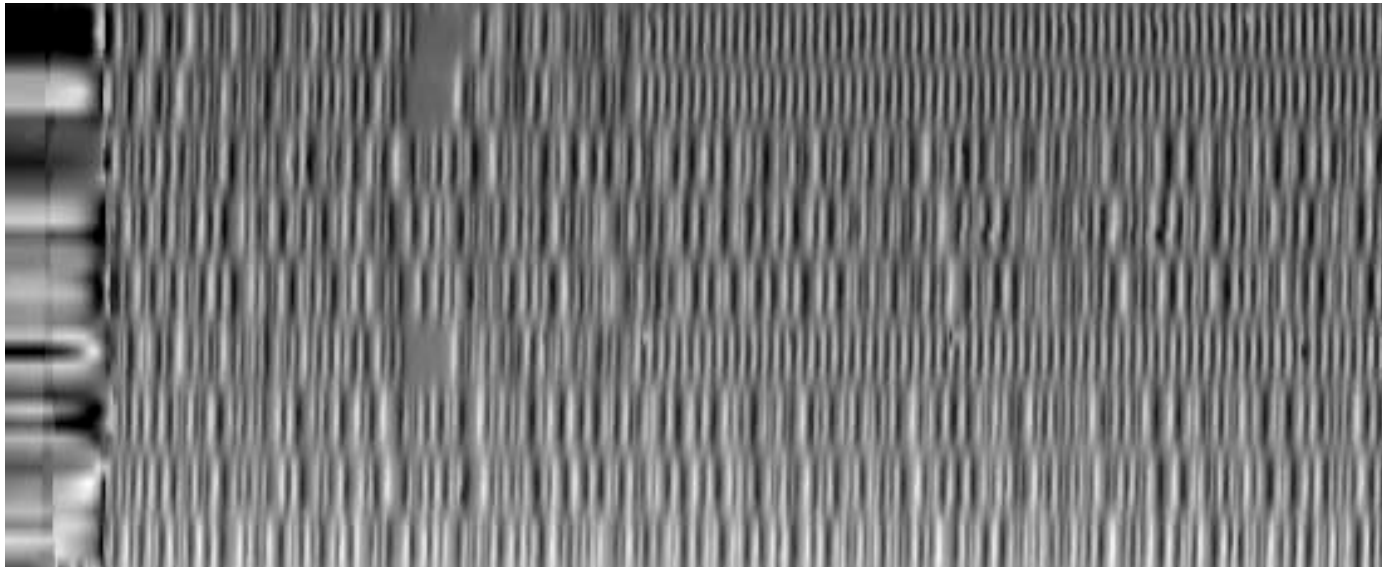


Hard Disk Drive – Magnetic Transitions



Linear Density: 135.4 kb/in or 5.33bits/ μm

32 μm



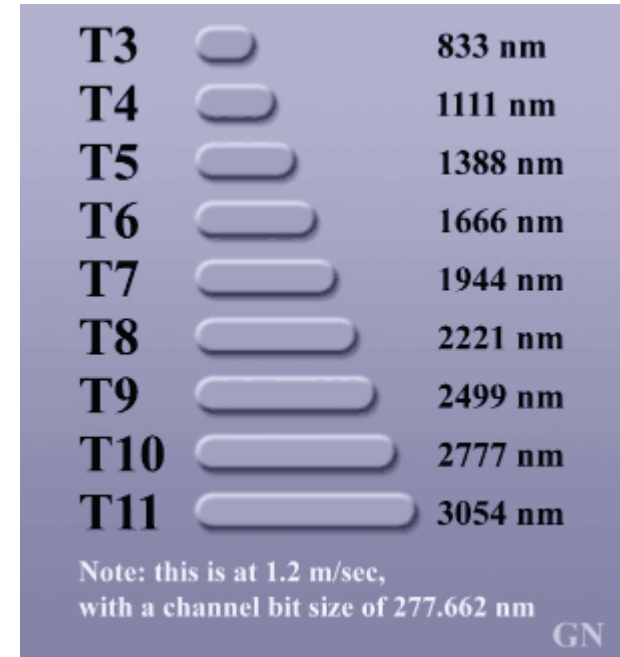
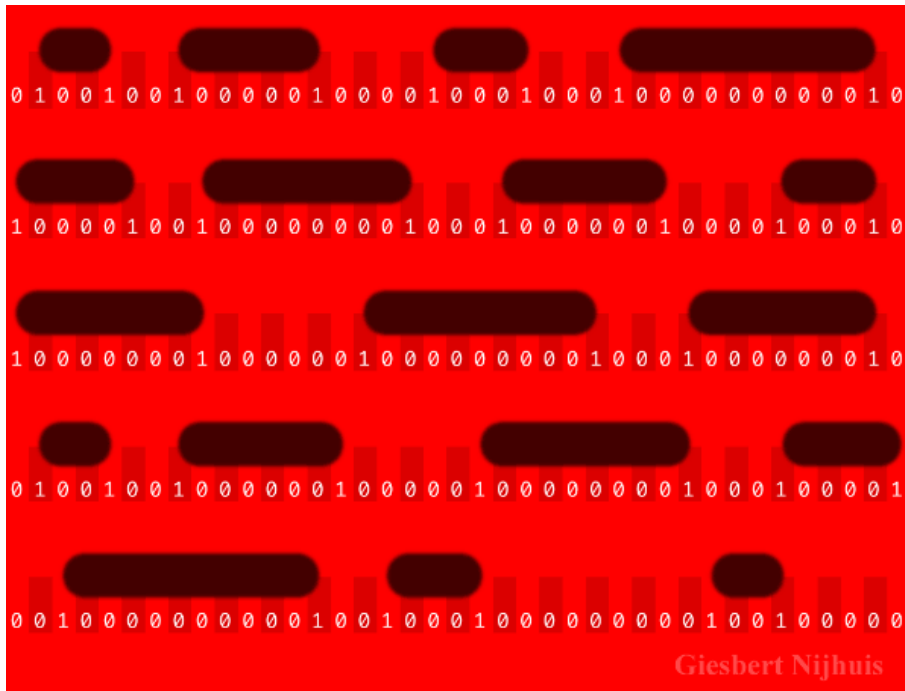
78 μm

Courtesy of Fred Spada

CD - Pits and Lands



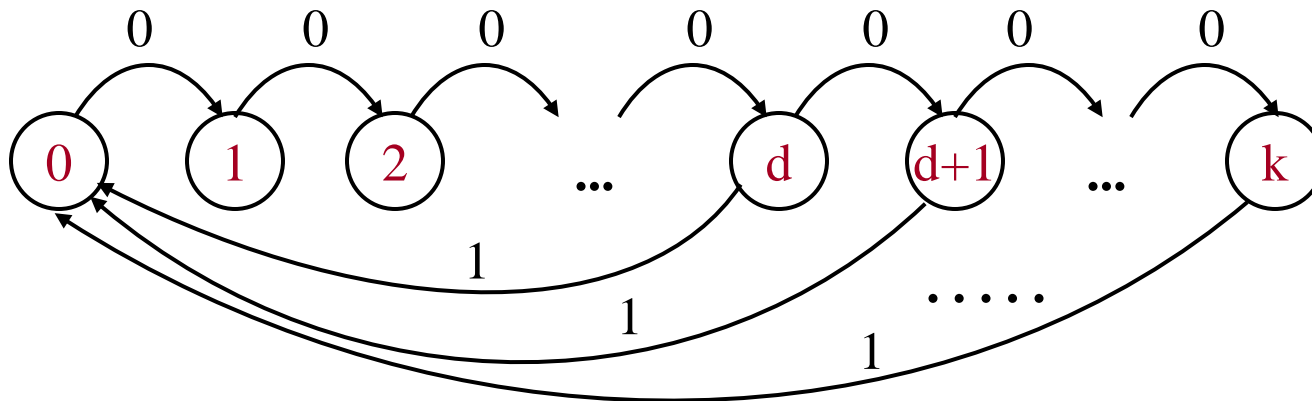
Linear Density: 39.4 kb/in or 1.55 bits/ μm



Courtesy of Giesbert Nijhuis

(d,k) Runlength-Limited Constraints

- Modulation codes for digital recording channels
 - d = minimum number of 0's between 1's
 - k = maximum number of 0's between 1's



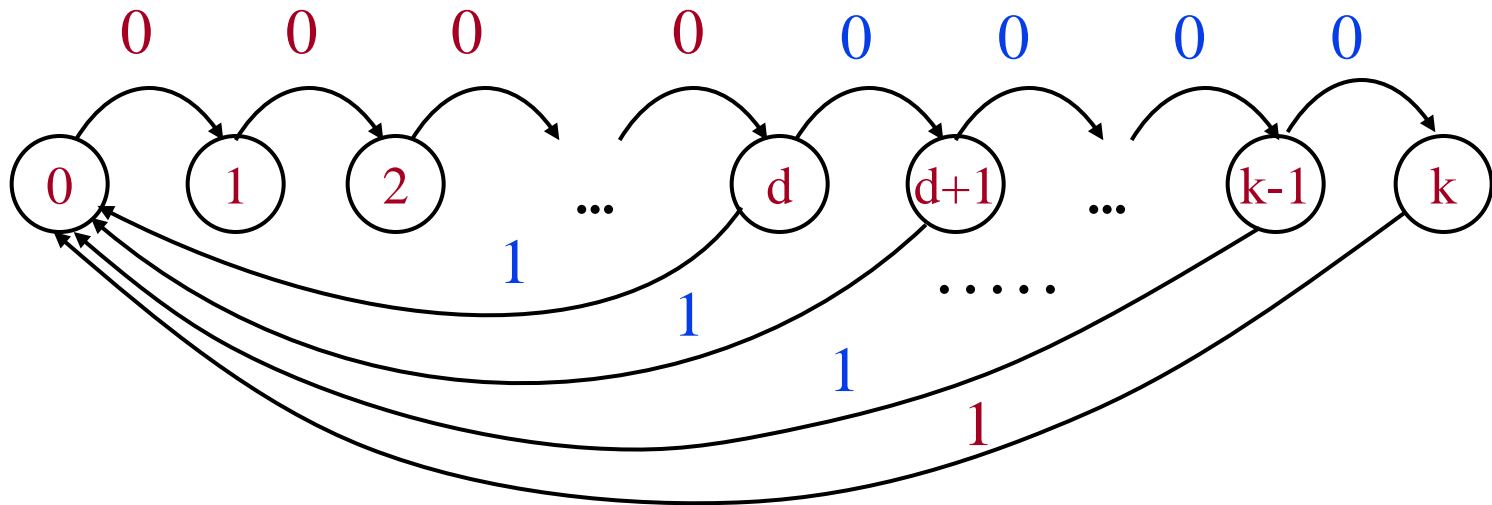
- CD uses $(d,k)=(2,10)$; Disk Drive uses $(d,k)=(0,4)$

Constrained Coding

- Problem:
 - How can we transform unconstrained binary data streams into (d,k) constrained binary code streams?
- Issues:
 - Invertibility of transformation (unique decodability)
 - Rate R , i.e., average ratio of # data bits to # code bits
 - Complexity of encoding and decoding operations

Bit-Stuffing Encoder

- Encoder “stuffs” extra bits into data stream, as needed to enforce the (d,k) constraint:
 - Stuffs d 0’s after every data 1
 - Keeps track of total number of consecutive 0’s
 - Stuffs a 1 and d 0’s after a runlength of k 0’s



Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence:

Decoder

Code sequence:

Data sequence:

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1

Decoder

Code sequence:

Data sequence:

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0

Decoder

Code sequence:

Data sequence:

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1

Decoder

Code sequence:

Data sequence:

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0

Decoder

Code sequence:

Data sequence:

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0

Decoder

Code sequence:

Data sequence:

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0

Decoder

Code sequence:

Data sequence:

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0

Decoder

Code sequence:

Data sequence:

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0

Decoder

Code sequence:

Data sequence:

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0

Decoder

Code sequence:

Data sequence:

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0

Decoder

Code sequence:

Data sequence:

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1

Decoder

Code sequence:

Data sequence:

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0

Decoder

Code sequence:

Data sequence:

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0

Decoder

Code sequence:

Data sequence:

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1

Decoder

Code sequence:

Data sequence:

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence:

Data sequence:

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence:

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1 0

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1 0

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1 0 0

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1 0 0 0

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1 0 0 0 1

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1 0 0 0 1

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1 0 0 0 1 0

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1 0 0 0 1 0

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1 0 0 0 1 0 0

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1 0 0 0 1 0 0 0

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1 0 0 0 1 0 0 0 1

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1 0 0 0 1 0 0 0 1

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1 0 0 0 1 0 0 0 1 0

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1 0 0 0 1 0 0 0 1 0

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Bit-Stuffing Demonstration

$(d,k)=(1,3)$

Encoder

Data sequence: 1 1 0 0 0 0 1 0 1

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Decoder

Code sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Data sequence: 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0

Biased Bits May Be Better!

- For large values of $k-d$, it seems like bit-stuffing may be more efficient if the data stream is properly biased, with fewer 1's than 0's, since 1's always generate d stuffed 0's.
- How can we transform a sequence of independent **unbiased** (i.e., fair) coin flips, where

$$\Pr(0) = \Pr(1) = 1/2$$

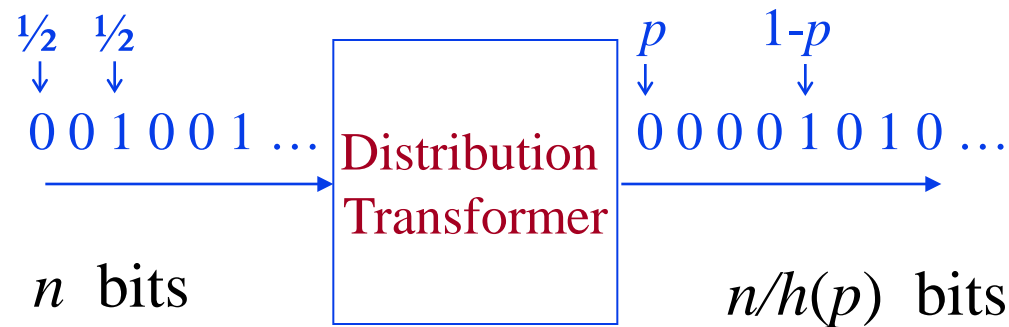
into a sequence of independent **biased** (i.e., not fair) coin flips, where, for $p \neq 1/2$

$$\Pr(0) = p$$

$$\Pr(1) = 1 - p$$

Distribution Transformer

- A “distribution transformer” maps an unbiased data stream to a biased stream invertibly:

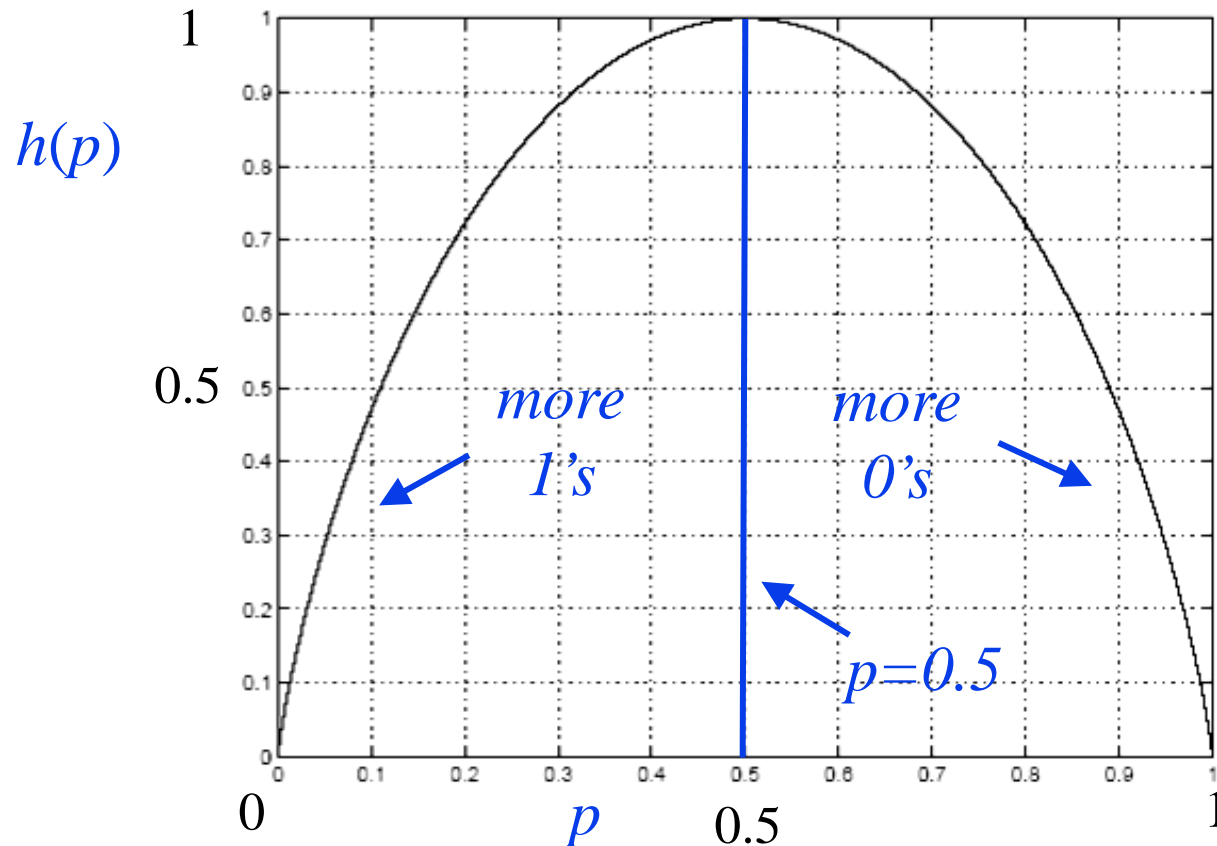


- There is a rate penalty, given by the binary entropy function:

$$h(p) = -p \log p - (1-p) \log (1-p) \leq 1$$

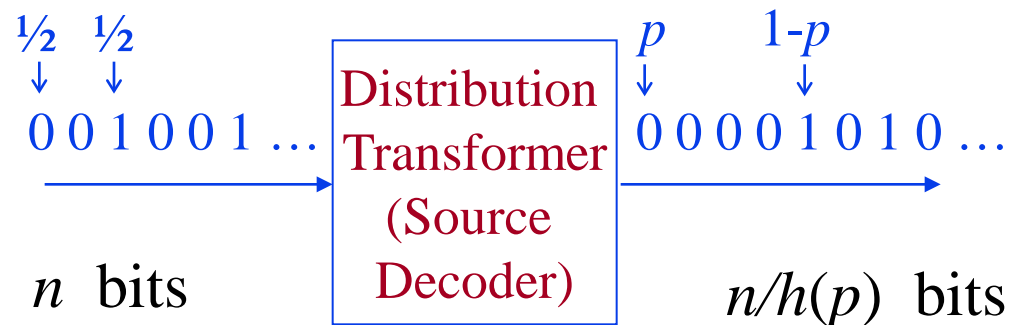
Binary Entropy Function

$$h(p) = -p \log p - (1-p) \log (1-p)$$



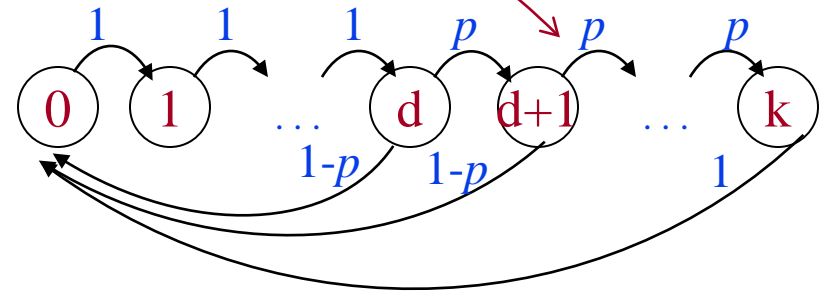
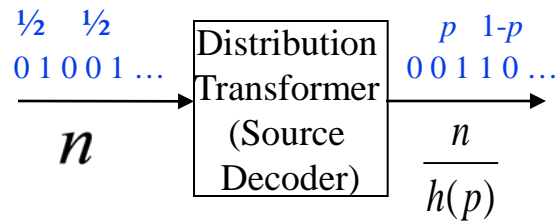
Distribution Transformer Implementation

- A “distribution transformer” can be implemented by using the decoder of a source code that compresses a p -biased sequence of bits with compression ratio $1/h(p)$ to 1.
- In practice, the transformer could be based upon stream-oriented arithmetic coding techniques and achieve a rate very close to $h(p)$.

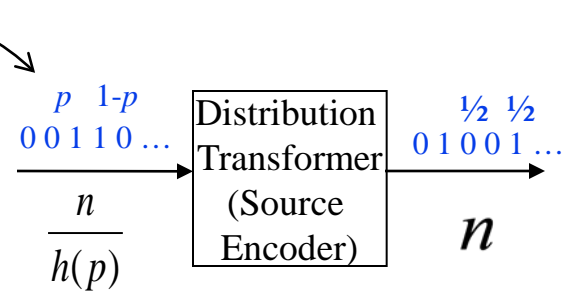
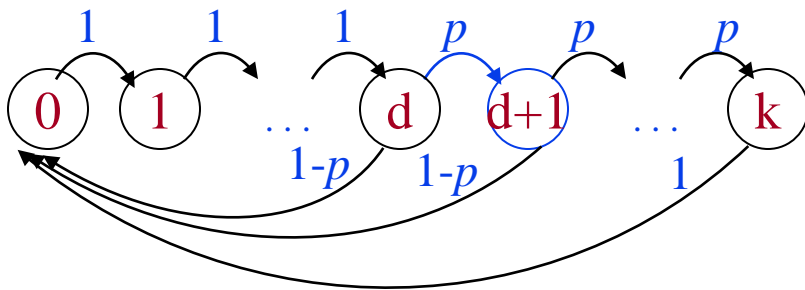


Bit-Stuffing Algorithm Flow

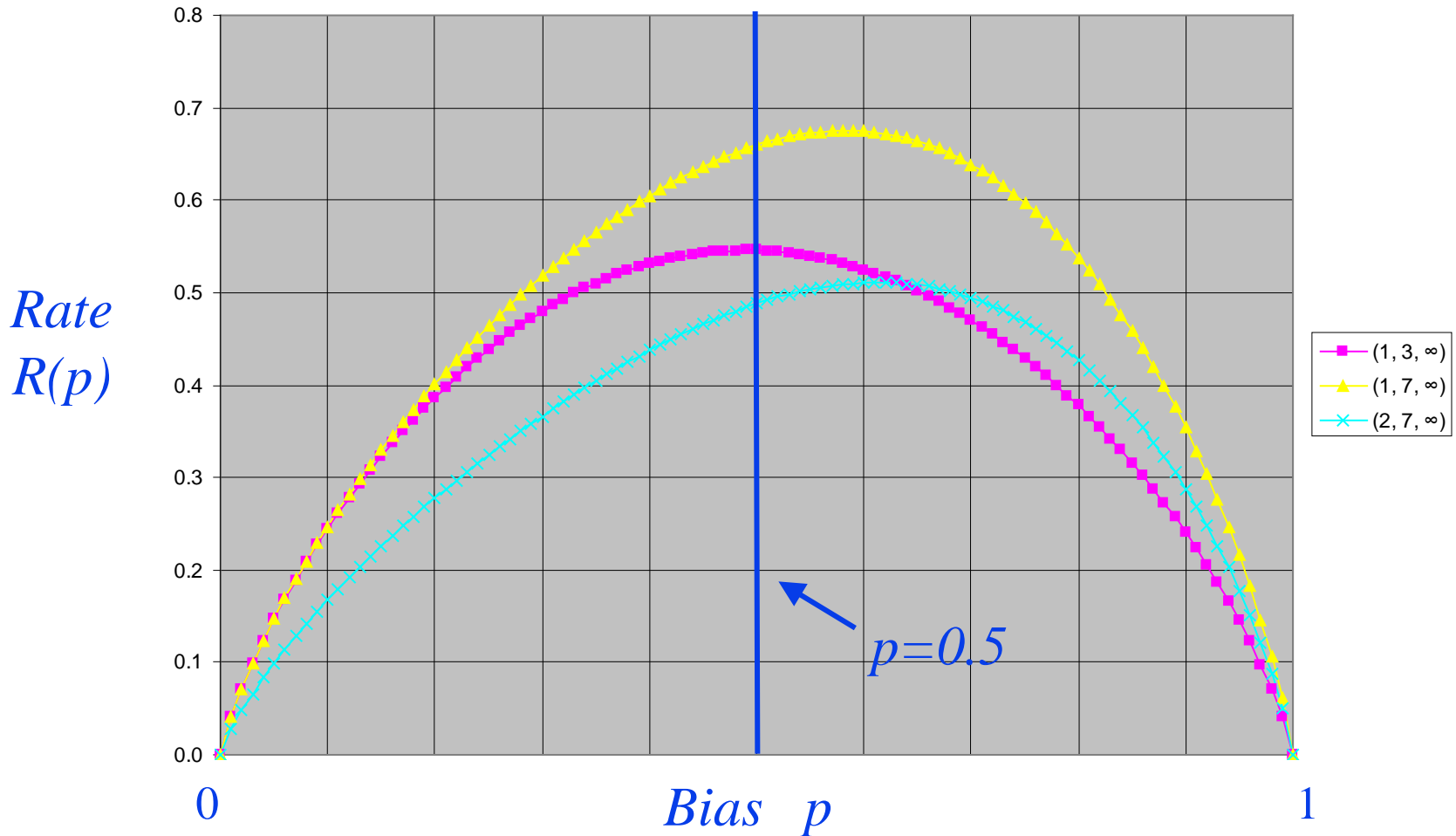
Encoder



Decoder

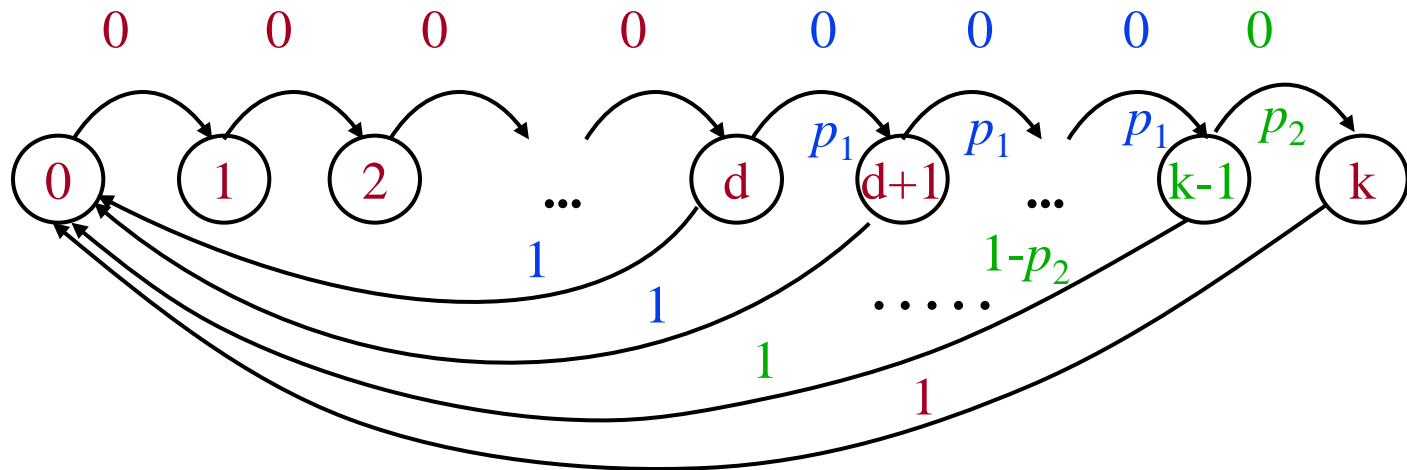


Bit-Stuffing Rate vs. Bias



Bit-Flipping

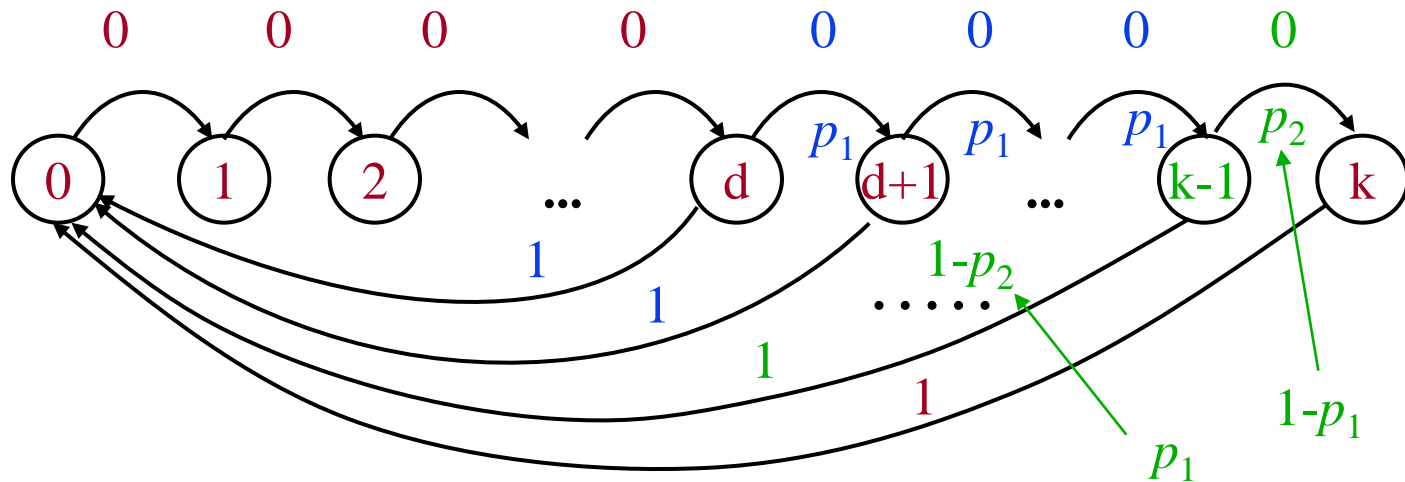
- After $k-1$ consecutive 0's, a 1 seems preferable to another 0: 1 generates d stuffed 0's, whereas a 0 generates a stuffed 1 along with d stuffed 0's.



- No need for a second distribution transformer if we just complement the next biased bit at state $k-1$, i.e. $p_2=1-p_1$

Bit-Flipping

- After $k-1$ consecutive 0's, a 1 seems preferable to another 0: 1 generates d stuffed 0's, whereas a 0 generates a stuffed 1 along with d stuffed 0's.



- No need for a second distribution transformer if we just “flip” the next biased bit at state $k-1$, i.e. $p_2=1-p_1$

Bit-Flipping vs. Bit-Stuffing

- **Theorem** [Aviran et al., 2003]: Bit-flipping achieves average rate **strictly higher** than bit-stuffing for $d \geq 1$ and $d+2 \leq k < \infty$.
(Also, starting the bit-flipping at state $k-1$ is optimal.)
- **Question:** Can we improve on bit-flipping, at least for some (d,k) constraints, still using only one distribution transformer?
YES! (e.g., symbol-sliding [Yogesh S. et al., 2004])

Other Natural Questions

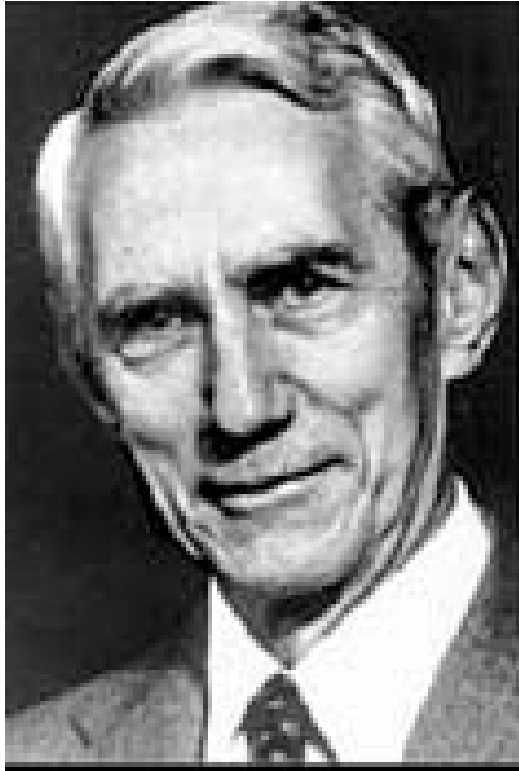
- Can we determine the absolute highest possible coding rate of (d,k) codes?

YES! [Shannon, 1948]

- Can we achieve it (or get arbitrarily close) using stuffing, flipping, sliding, multiple transformers, or other coding methods?

YES! [Shannon, 1948] , and his followers...

Claude E. Shannon



Claude Elwood Shannon
1916 - 2001



Shannon Statue – CMRR

The Inscription on the Statue

CLAUDE ELWOOD SHANNON

1916 – 2001

FATHER OF INFORMATION THEORY

**HIS FORMULATION OF THE MATHEMATICAL
THEORY OF COMMUNICATION PROVIDED
THE FOUNDATION FOR THE DEVELOPMENT OF
DATA STORAGE AND TRANSMISSION SYSTEMS
THAT LAUNCHED THE INFORMATION AGE.**

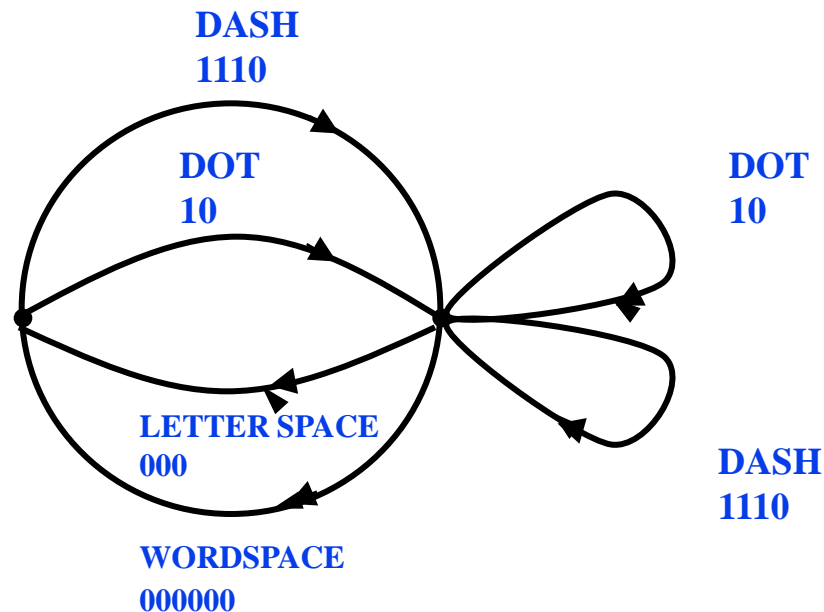
DEDICATED OCTOBER 16, 2001

EUGENE DAUB, SCULPTOR

Discrete Noiseless Channels (Constrained Systems)

- A constrained system S is the set of sequences generated by walks on a labeled, directed graph G .

Telegraph channel constraints [Shannon, 1948]



Constrained Codes and Capacity

- Shannon showed that the number of length- n constrained sequences is approximately 2^{Cn} .
- The quantity C is called the capacity of the constrained system.

Theorem [Shannon,1948] : If there exists a decodable code at rate $R = m/n$ from binary data to S , then $R \leq C$.

Theorem [Shannon,1948] : For any rate $R = m/n < C$ there exists a block code [look-up table] from binary data to S with rate $km:kn$, for some integer $k \geq 1$.

Computing Capacity

- Shannon also showed how to compute the capacity C .
- For (d,k) constraints, $C_{d,k} = \log \lambda_{d,k}$, where $\lambda_{d,k}$ is the largest real root of the polynomial

$$f_{d,k}(x) = x^{k+1} - x^{k-d} - \dots - x - 1, \text{ for } k < \infty$$

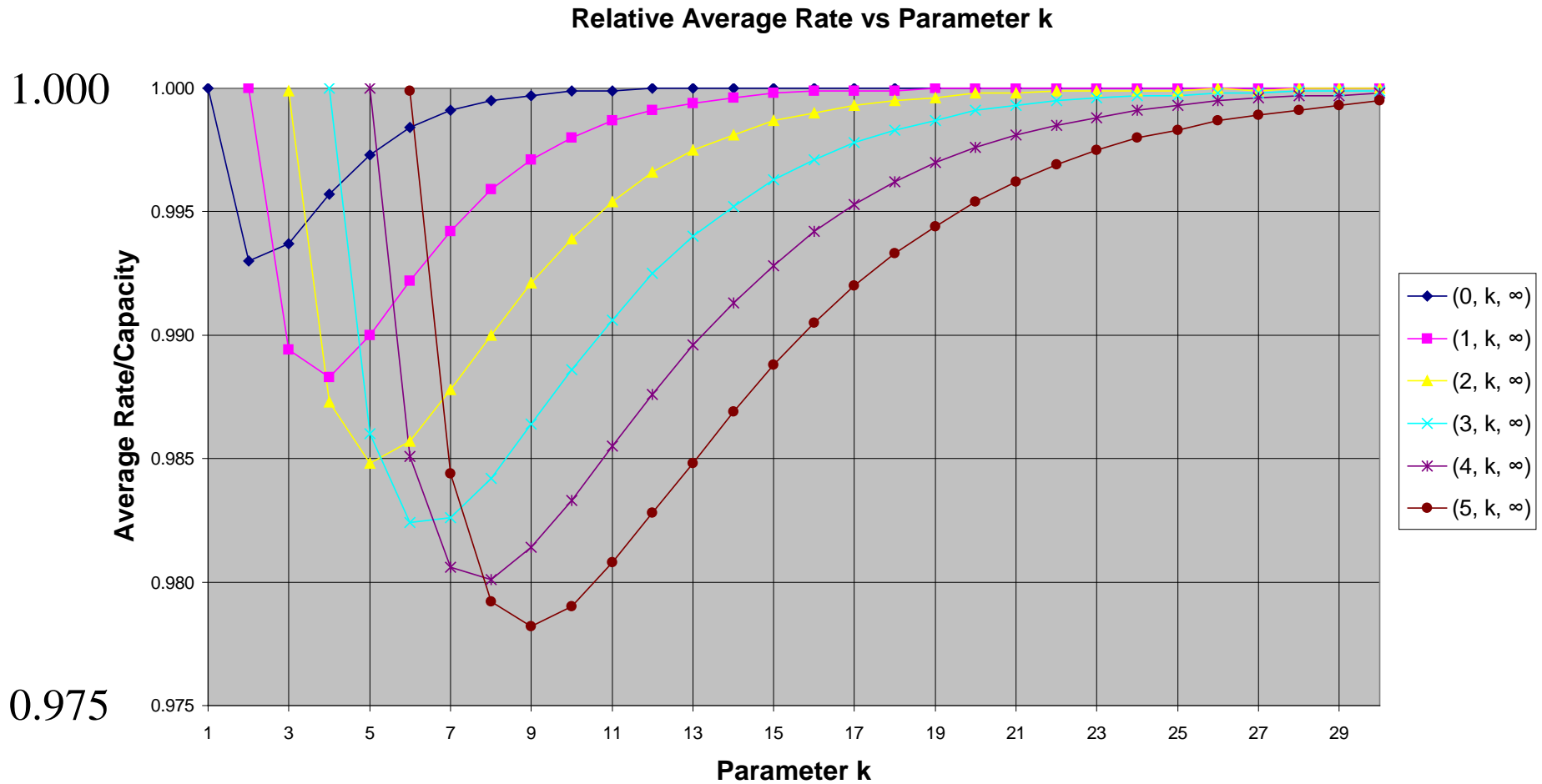
- For (d,∞) constraints, use the relation [Ashley et al. 1987]:

$$C_{d,\infty} = C_{d-1,2d-1}, \text{ for } d \geq 1.$$

Achieving Capacity (sometimes...)

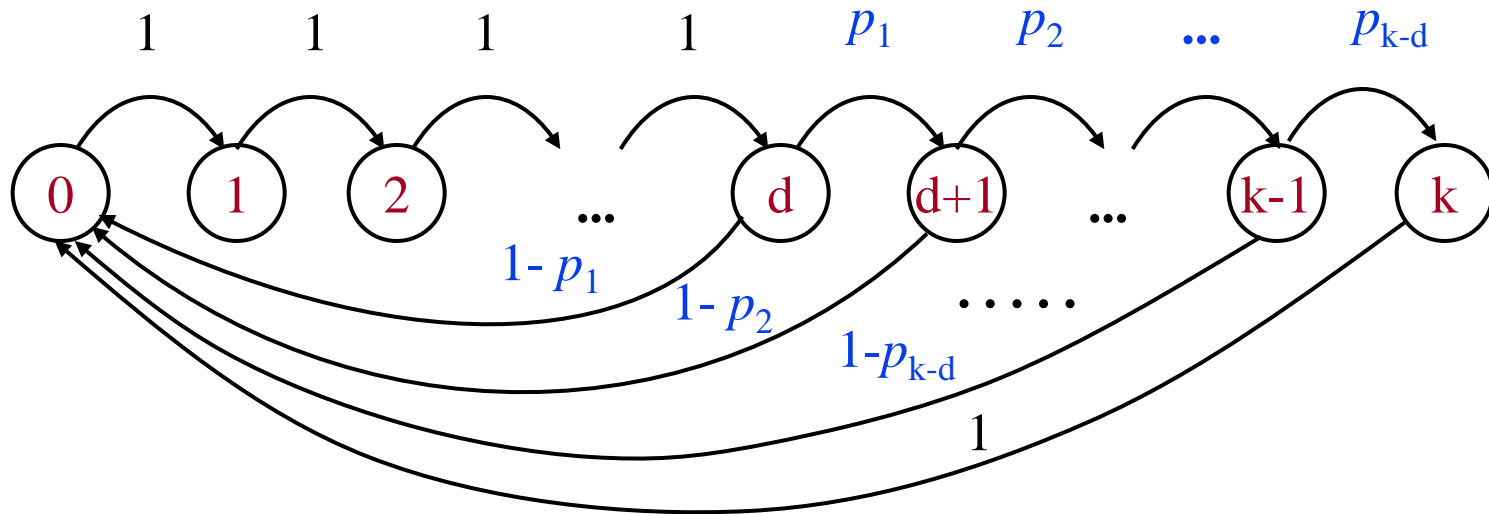
- **Theorem [Bender-Wolf, 1993]:** The bit-stuffing algorithm achieves capacity for $(d,k)=(d,d+1)$ and $(d,k)=(d,\infty)$.
- **Theorem [Aviran, et al., 2004]:** The bit-flipping algorithm additionally achieves capacity for $(d,k)=(2,4)$.
- **Theorem [Yogesh S.-McLaughlin]:** The symbol-sliding algorithm also achieves capacity for $(d,k)=(d,2d+1)$.

Bit Stuffing Performance

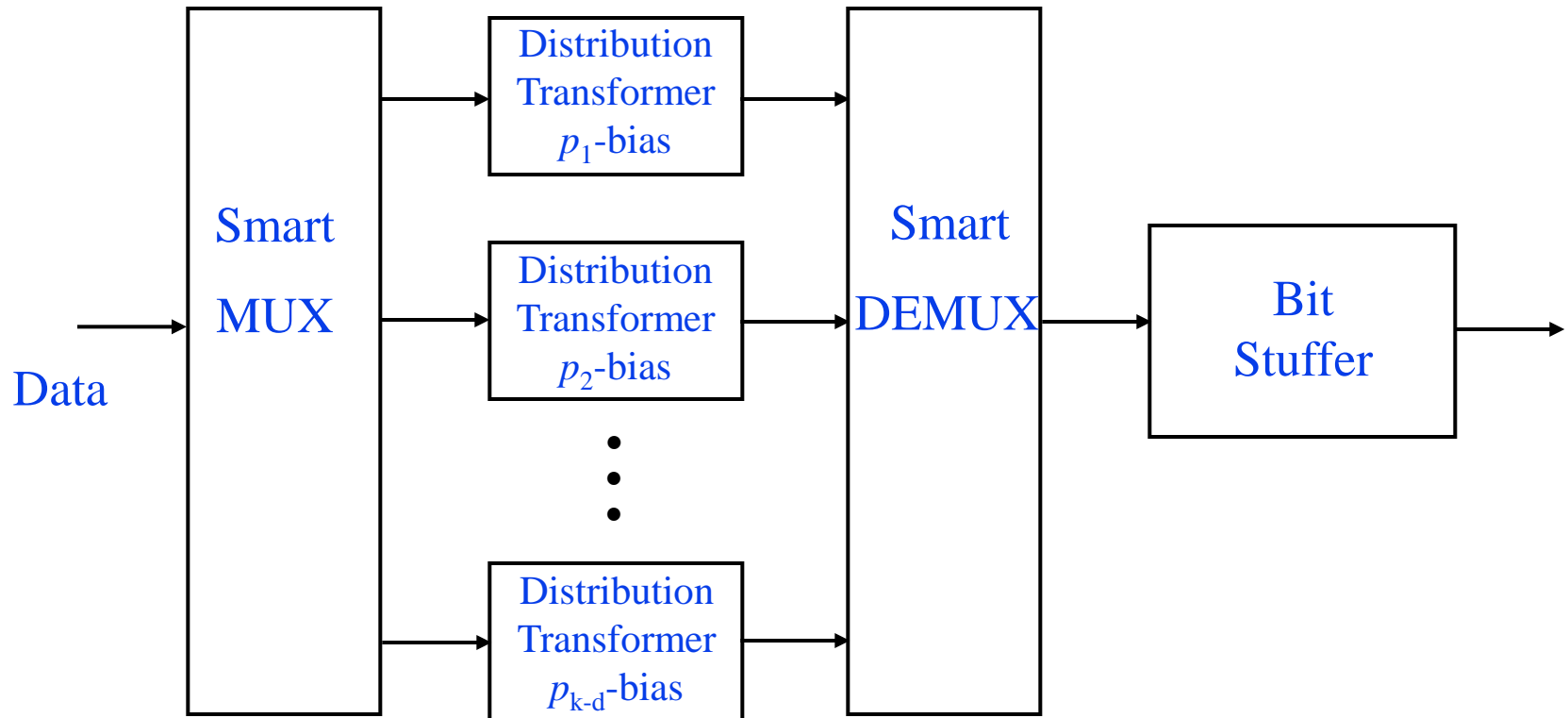


Shannon Probabilities

- Shannon also determined the probabilities on the edges of the constraint graph that correspond to the highest achievable rate, i.e., that **achieve capacity**, in terms of λ .
- The constrained sequences are called “maxentropic.”



Bit-Stuffing with Multiple Transformers



- **Maxentropic encoder achieves capacity!**

The Formula on the “Paper”

Capacity of a discrete channel with noise [Shannon, 1948]

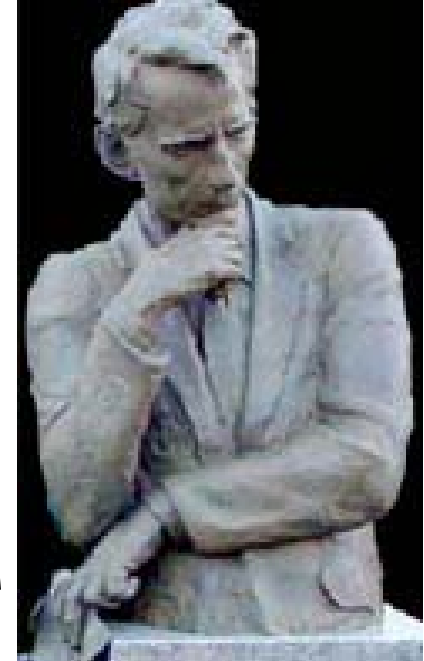
$$C = \text{Max} (H(x) - H_y(x))$$

For noiseless channel, $H_y(x)=0$, so:

$$C = \text{Max} H(x)$$

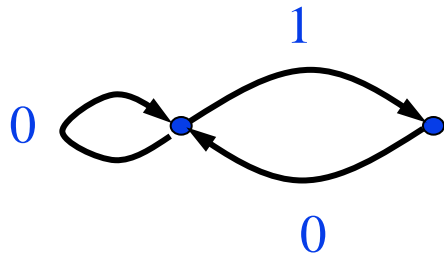


Capacity achieved by maximum entropy sequences

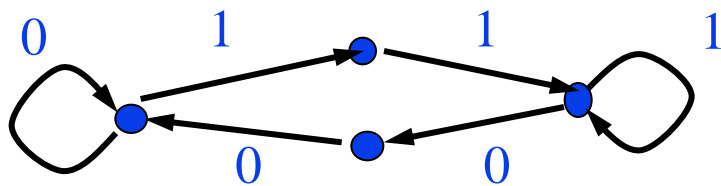


Magnetic Recording Constraints

Runlength constraints
 (“finite-type”: determined by finite list F of forbidden words)



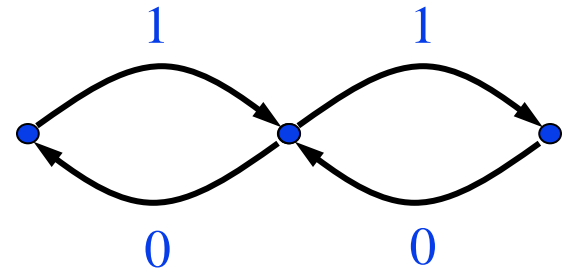
Forbidden word $F = \{11\}$



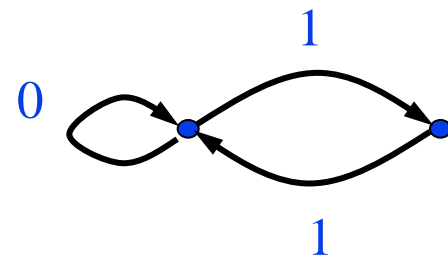
Forbidden words $F = \{101, 010\}$

Spectral null constraints
 (“almost-finite-type”)

Biphase



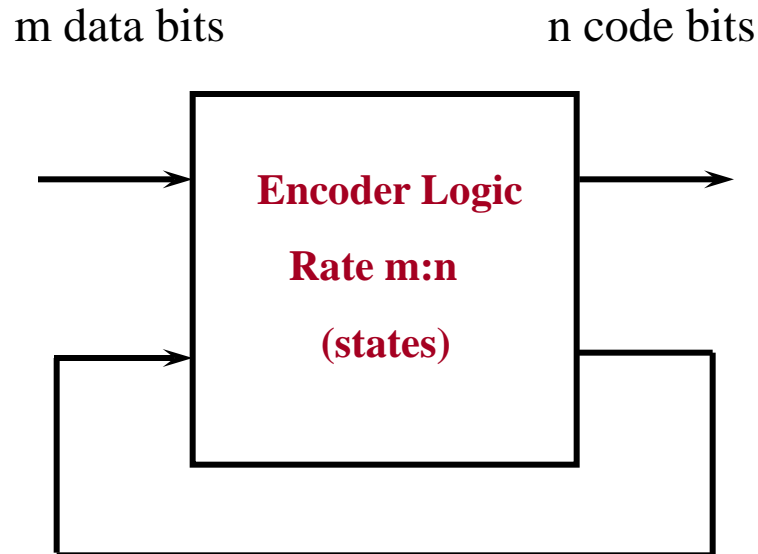
Even



Practical Constrained Codes

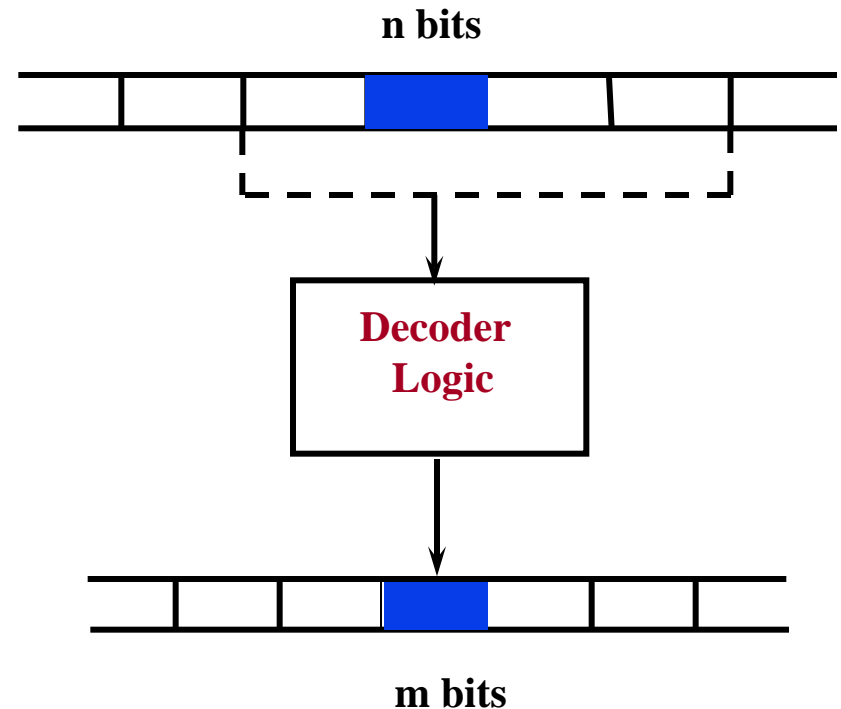
Finite-state encoder

(from binary data into S)



Sliding-block decoder

(inverse mapping from S to data)



We want: high rate $R=m/n$
low complexity

Constrained Coding Theorems

- Powerful coding theorems were motivated by the problem of constrained code design for magnetic recording.

Theorem[Adler-Coppersmith-Hassner, 1983]

Let S be a finite-type constrained system. If $m/n \leq C$, then there exists a rate $m:n$ sliding-block decodable, finite-state encoder.

(Proof is constructive: “state-splitting algorithm.”)

Theorem[Karabed-Marcus, 1988]

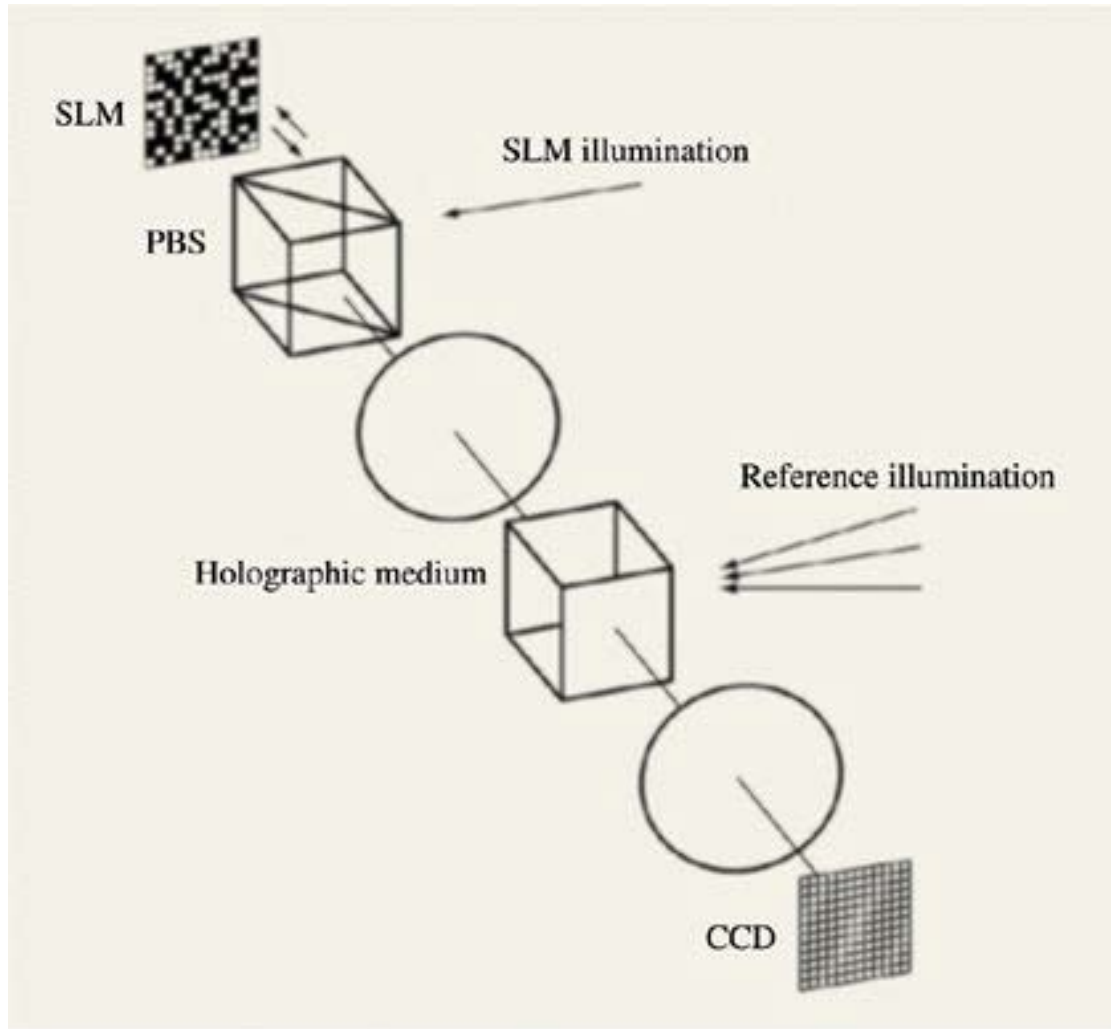
Ditto if S is almost-finite-type.

(Proof not so constructive...)

Two-Dimensional Constrained Systems

- “Page-oriented” and “multi-track” recording technologies require 2-dimensional (2-D) constraints.
- Examples:
 - **Holographic Storage - InPhaseTechnologies**
 - **Two-Dimensional Optical Storage (TwoDOS) – Philips**
 - **Patterned Magnetic Media – Hitachi, Toshiba, ...**
 - **Thermo-Mechanical Probe Array – IBM**

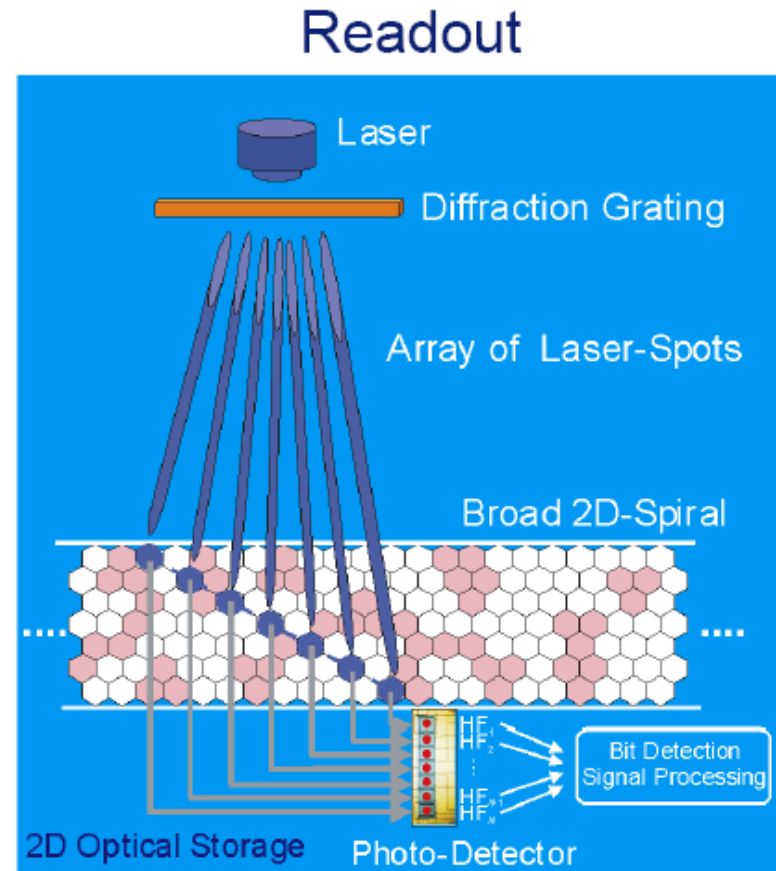
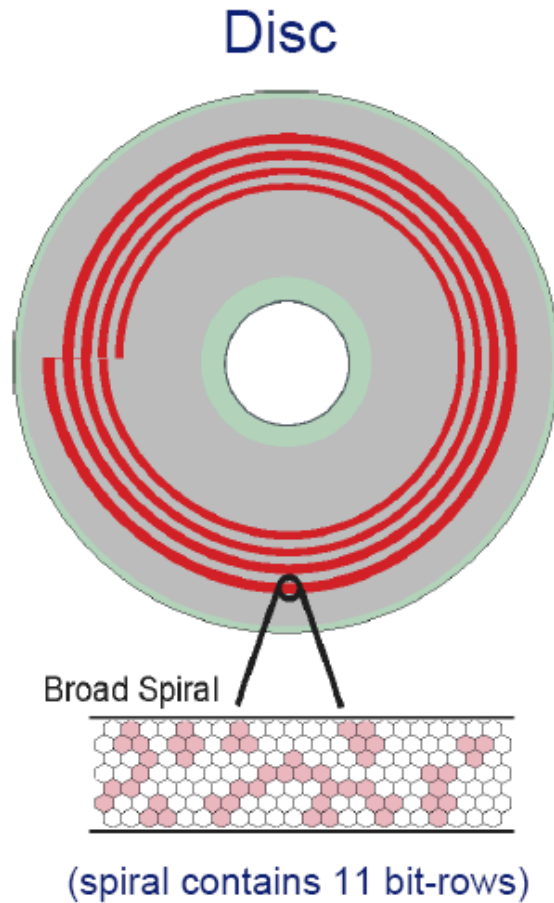
Holographic Recording



Array constraints:

- 2-D runlength limited
- 2-D non-isolated bit
- 2-D low-pass filter

TwoDOS



Courtesy of Wim Coene, Philips Research

2-D Constrained Codes

- There is no comprehensive algorithmic theory for constructing encoders and decoders for 2-D constrained systems. (See, however, [Demirkan-Wolf, 2004] .)
- There is no known general method for computing the capacity of 2-D constraints.
- That being the case, let's try...

2-D bit-stuffing!

Constraints on the Integer Grid

- 2-D (d,k) constraints satisfy the (d,k) constraint in rows and columns.

1	0	0	0	1	0	0
0	0	0	1	0	1	0
0	1	0	0	1	0	0
1	0	1	0	0	0	1
0	1	0	1	0	1	0
0	0	1	0	1	0	0
1	0	0	1	0	0	0

- $(d,k) = (1, \infty)$ constraint in rows and columns.

“Hard-Square Model”

2-D Bit-Stuffing Encoder (Hard-Square Model)

- Biased sequence: 1 1 1 0 0 0 1 0 0 1 0 0 0 0 1 1 0 0 0

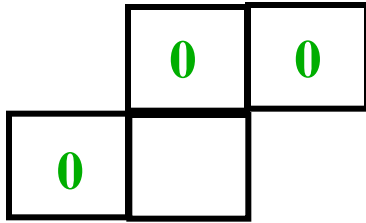
1	0	1	0	0	0	0			
0	1	0	1	0	0				
0	0	0	0	1	0				
0	0	0	1	0					
1	0	0	0						
0	0								
0									

Optimal bias $\Pr(0) = p = 0.6444$

$$*R(p) = 0.583056*$$

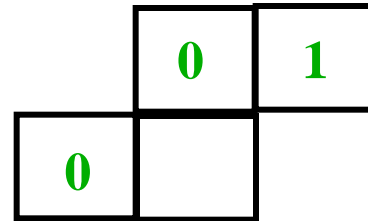
Enhanced Bit-Stuffing Encoder (Hard-Square Model)

- Use 2 source encoders, with parameters p_0, p_1 .



Optimal bias

$$\Pr(0) = p_0 = 0.671833$$



Optimal bias

$$\Pr(0) = p_1 = 0.566932$$

$$R(p_1, p_2) = 0.587277$$

Capacity of 2-D (d,k) Constraints

- For 2-D (d,k) constraints, there is no known simple “formula” that lets us compute the capacity $C^{d,k}$.
- In fact, the only nontrivial (d,k) pairs for which $C^{d,k}$ is known are those with zero capacity [Ashley-Marcus 1998], [Kato-Zeger 1999]:

$$C^{d,d+1} = 0, \text{ for } d \geq 1$$

Capacity of Hard-Square Model

- Sharp bounds on $C^{1,\infty}$ have been computed:

$$\underline{0.587891161775} \leq C^{1,\infty} \leq \underline{0.587891161868}$$

[Calkin-Wilf, 1998] , [Nagy-Zeger, 2000]:

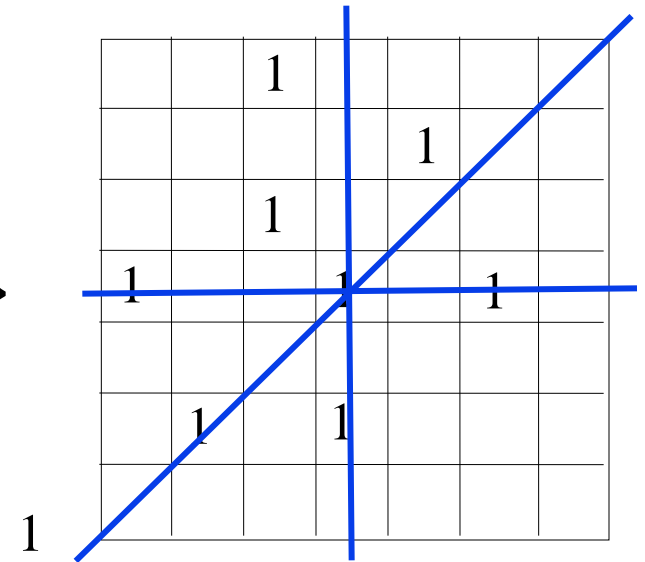
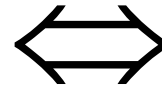
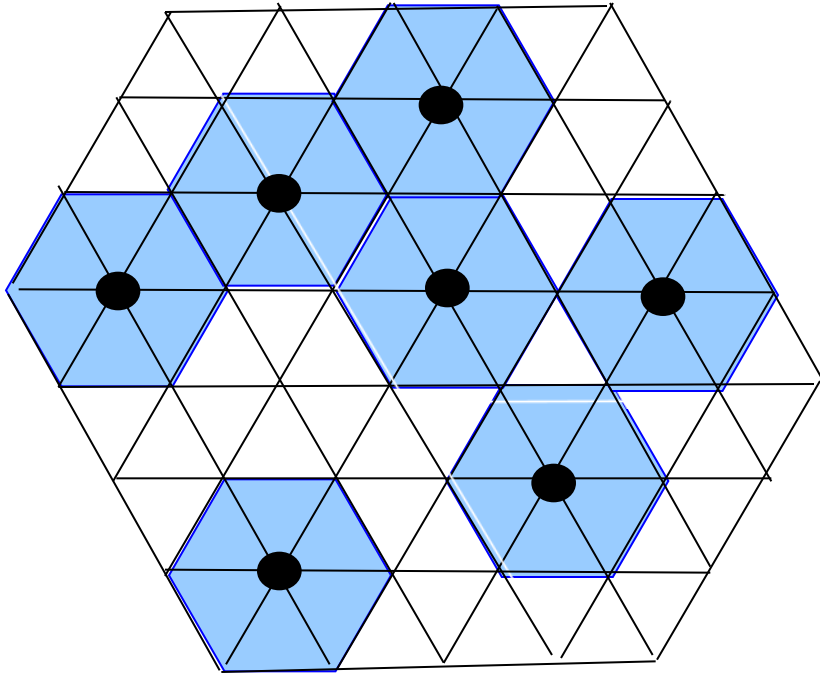
- Bit-stuffing encoder with one distribution transformer achieves rate $R(p) = 0.583056$ within 1% of capacity.
- Bit-stuffing encoder with two distribution transformers achieves rate $R(p_1, p_2) = 0.587277$ within 0.1% of capacity.

Bit-Stuffing Bounds on $C^{d,\infty}$

- Bit-stuffing encoders can also be extended to 2-D (d,∞) constraints.
- Bounds on the bit-stuffing encoder rate yield the best known lower bounds on $C^{d,\infty}$ for $d > 1$ [Halevy, et al., 2004].
- Bit-stuffing can also be applied to 2-D constraints on the hexagonal lattice.
- Bounds on the bit-stuffing encoder rate yield the best known lower bounds on $C^{d,\infty}_{hex}$ for $d > 1$ [Halevy, et al., 2004]

Hard Hexagon Model

- $(d,k)=(1,\infty)$ constraints on the hexagonal lattice



$$F = \left\{ \begin{bmatrix} 1 \\ 1 \end{bmatrix}, [1 \quad 1], \begin{bmatrix} & 1 \\ 1 & \end{bmatrix} \right\}$$

Hard-Hexagon Model

Hard Hexagon Capacity

- Capacity of hard hexagon model $C_{hex}^{1,\infty}$ is known precisely! [Baxter,1980]*

$C_{hex}^{1,\infty} = \log \kappa_h$, where $\kappa = \kappa_1 \kappa_2 \kappa_3 \kappa_4$ and

$$\kappa_1 = 4^{-1} 3^{5/4} 11^{-5/12} c^{-2}$$

$$\kappa_2 = \left[1 - \sqrt{1-c} + \sqrt{2+c+2\sqrt{1+c+c^2}} \right]^2$$

$$\kappa_3 = \left[-1 - \sqrt{1-c} + \sqrt{2+c+2\sqrt{1+c+c^2}} \right]^2$$

$$\kappa_4 = \left[\sqrt{1-a} + \sqrt{2+a+2\sqrt{1+a+a^2}} \right]^{-1/2}$$

$$a = -\frac{124}{363} 11^{1/3}$$

$$b = \frac{2501}{11979} 33^{1/2}$$

$$c = \left[\frac{1}{4} + \frac{3}{8} a \left[(b+1)^{1/3} - (b-1)^{1/3} \right] \right]^{1/3}$$

So,

$$C_{hex}^{1,\infty} \approx 0.480767622$$

Bit-stuffing encoder achieves rate within **0.5%** of capacity!

*Hard Hexagon Capacity**

- Alternatively, the hard hexagon entropy constant K satisfies a degree-24 polynomial with (big!) integer coefficients.
- Baxter does offer this disclaimer regarding his derivation, however:

*“It is not mathematically rigorous, in that certain analyticity properties of κ are assumed, and the results of Chapter 13 (which depend on assuming that various large-lattice limits can be interchanged) are used. However, I believe that these assumptions, and therefore (14.1.18)-(14.1.24), are in fact correct.”

Concluding Remarks

- The theory and practice of 1-D constrained coding, including bit-stuffing, is well-developed and powerful.
- The lack of convenient graph-based representations of 2-D constraints prevents the straightforward extension of 1-D techniques for information theoretic analysis and code design. Both are active research areas.
- Bit-stuffing encoders yield some of the best known bounds on capacity of 2-D constraints.
- There are connections to statistical physics that may open up new approaches to understanding 2-D constrained systems (and, perhaps, vice-versa).

Acknowledgments

- Thanks to my colleagues:
 - Ron Roth, Jack Wolf, Ken Zeger
- Thanks to our students:
 - Sharon Aviran, Jiangxin Chen, Shirley Halevy, Zsigmond Nagy
- Thanks to you – for listening (and bit-stuffing)!!