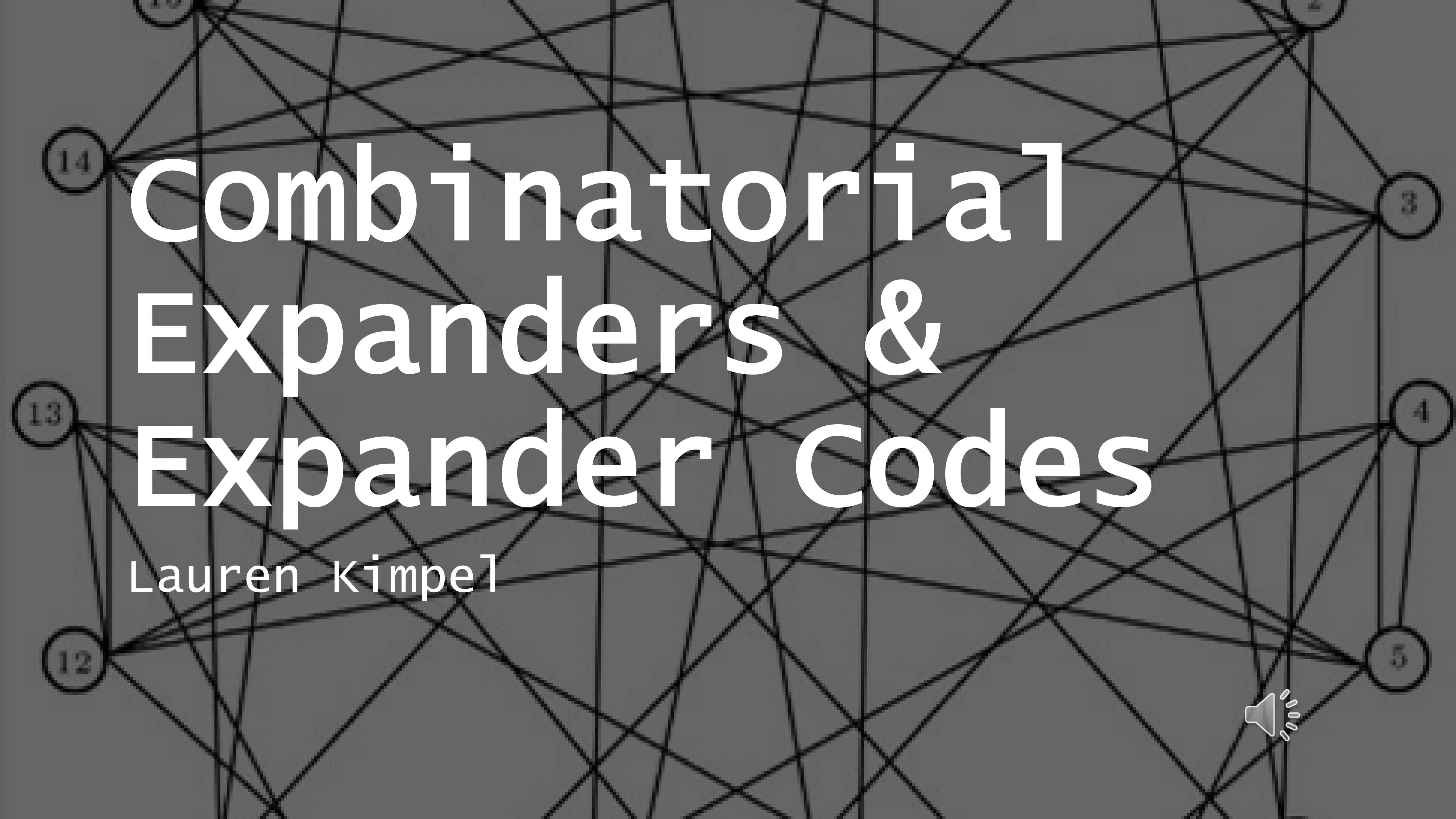
Combinatorial Expanders & Expander Codes

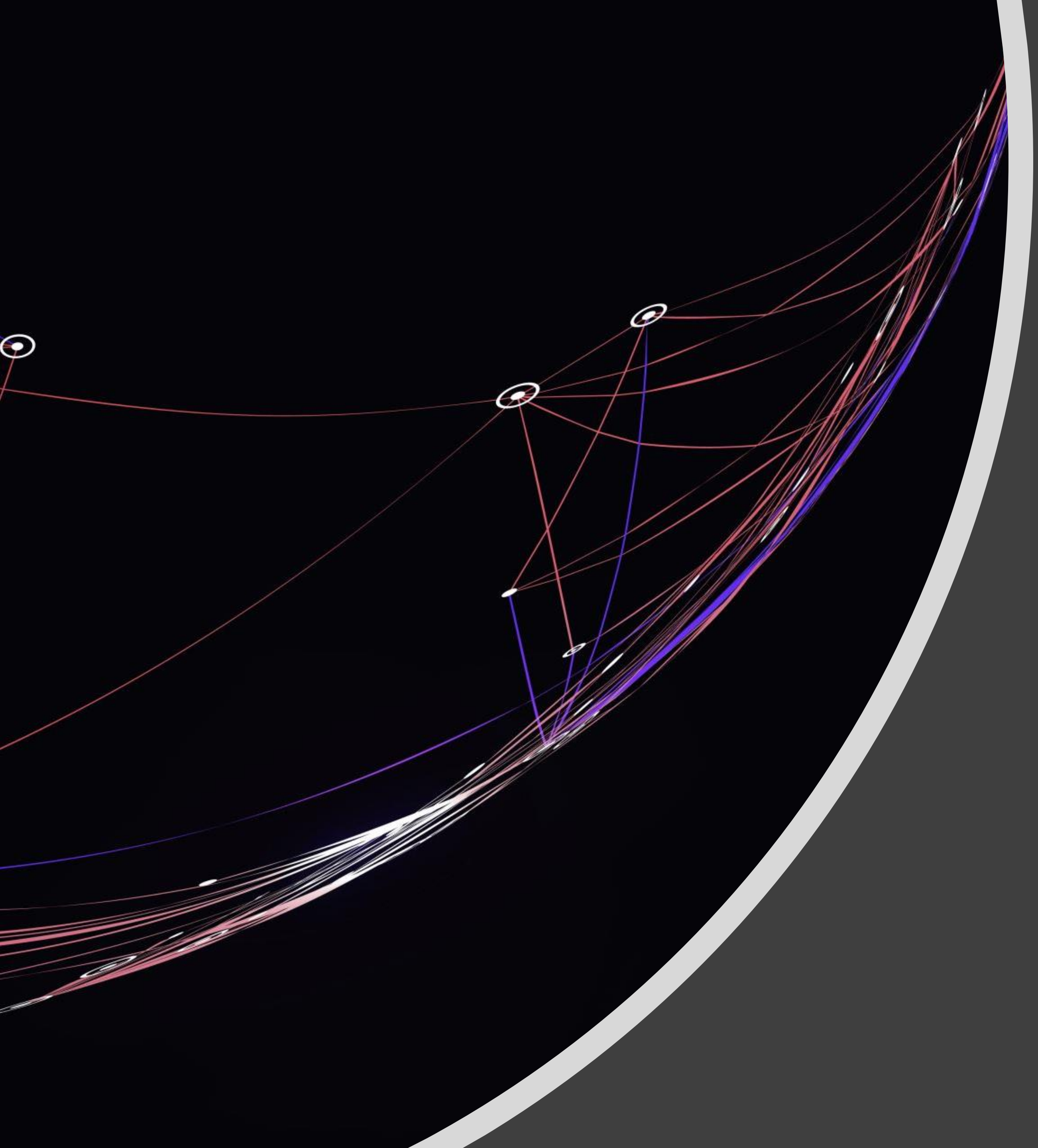
Lauren Kimpel



Part I: Theory

What'll Be Discussed

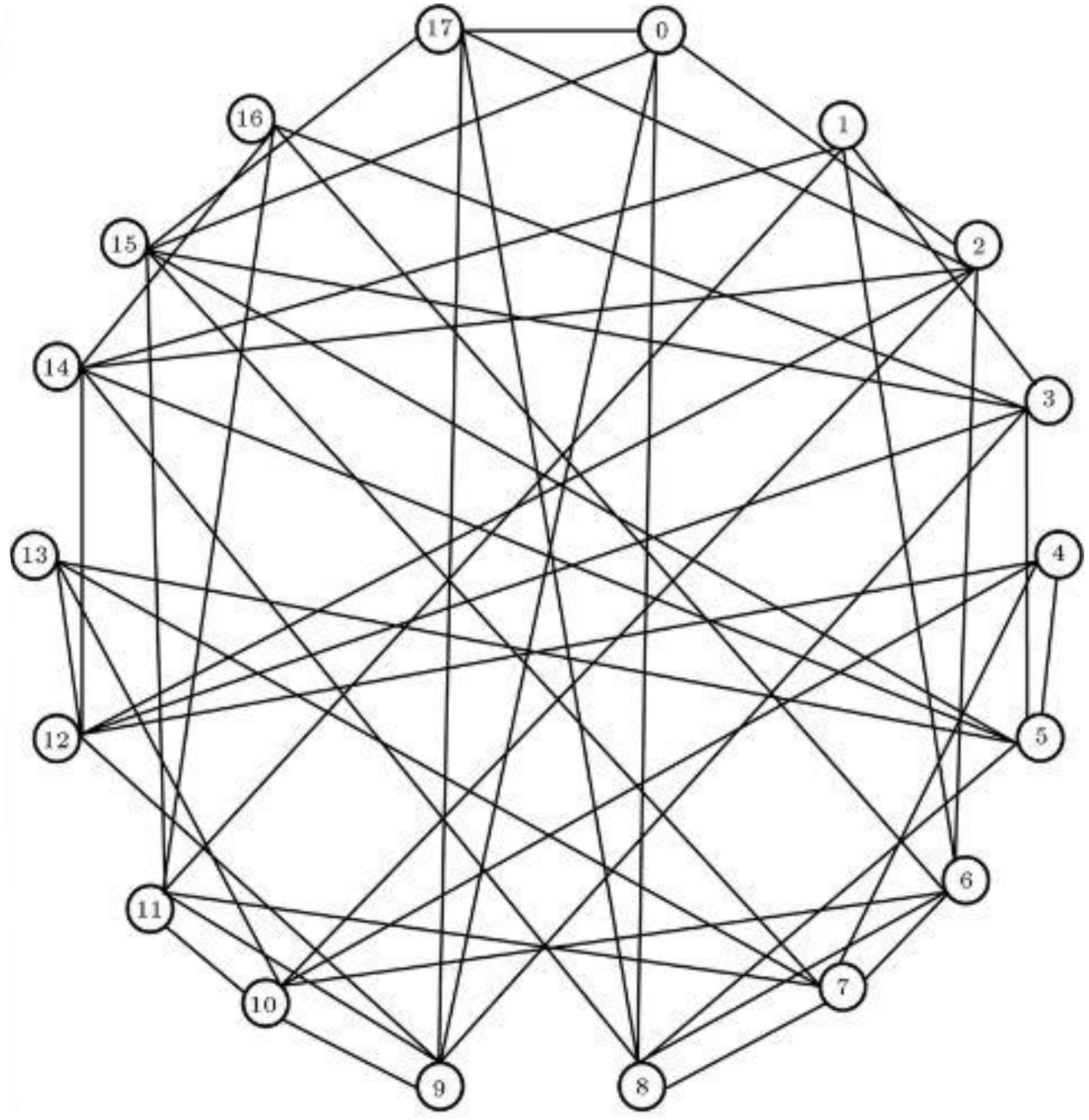
- Theory of Expander Graphs
  - How to Construct Them
    - Origins and Explicit Constructions
      - Zig-Zag Products
  - Edge constraints and combinatorial properties
- Information Theory Primer
  - General Vocab
  - Linear Codes & Examples
    - Parity Check and Generator Matrices
    - How to Tell the Quality of a Code
    - LPDC Codes
- Expander Code Advantages



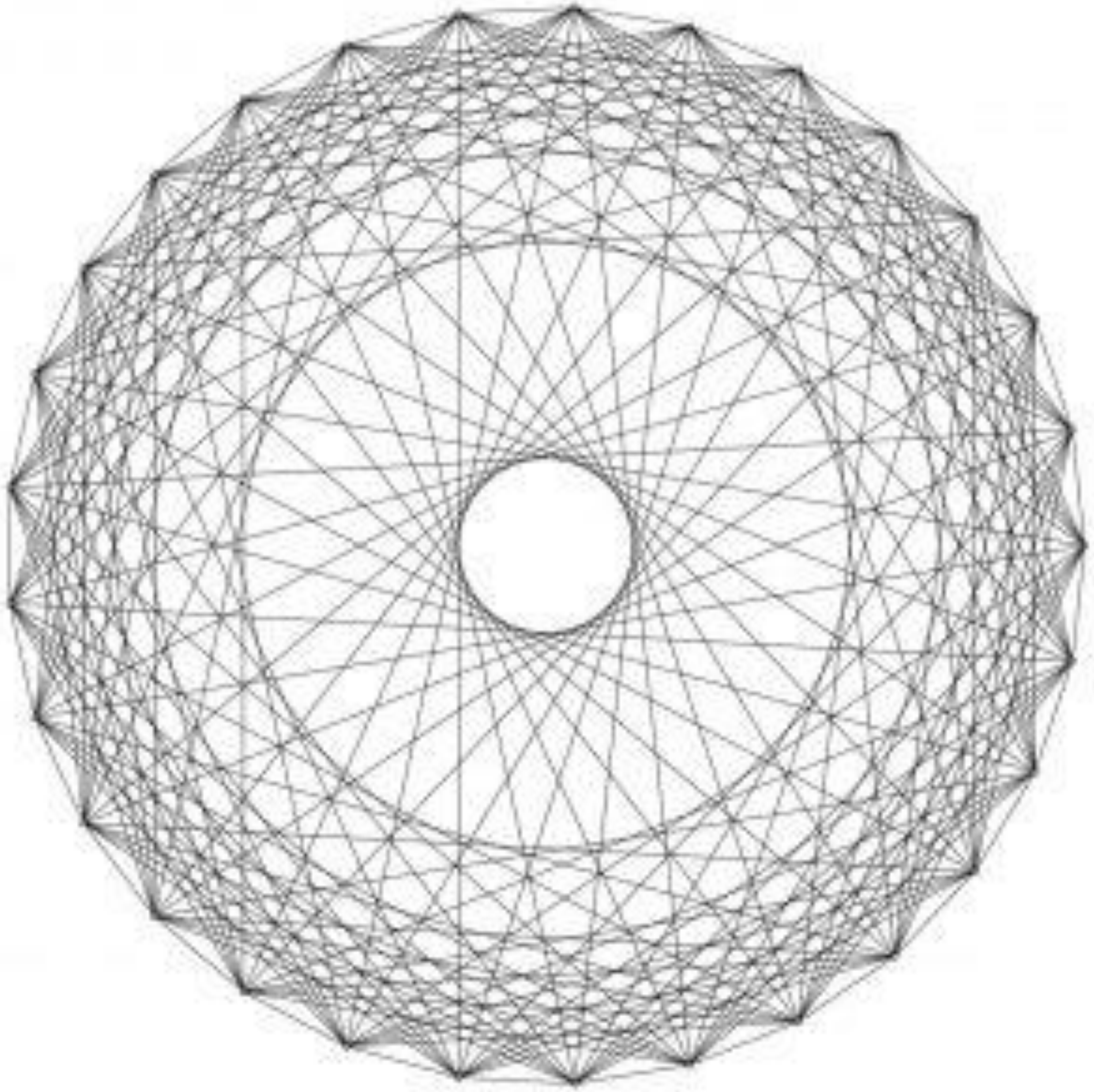
Why Are Expanders Interesting?

- These are highly-connected graphs, ideally very sparse
- Relevance to number theory, algebraic geometry and topology, information theory, quantum computing, cryptography, design and optimization of robust networks, social networks…

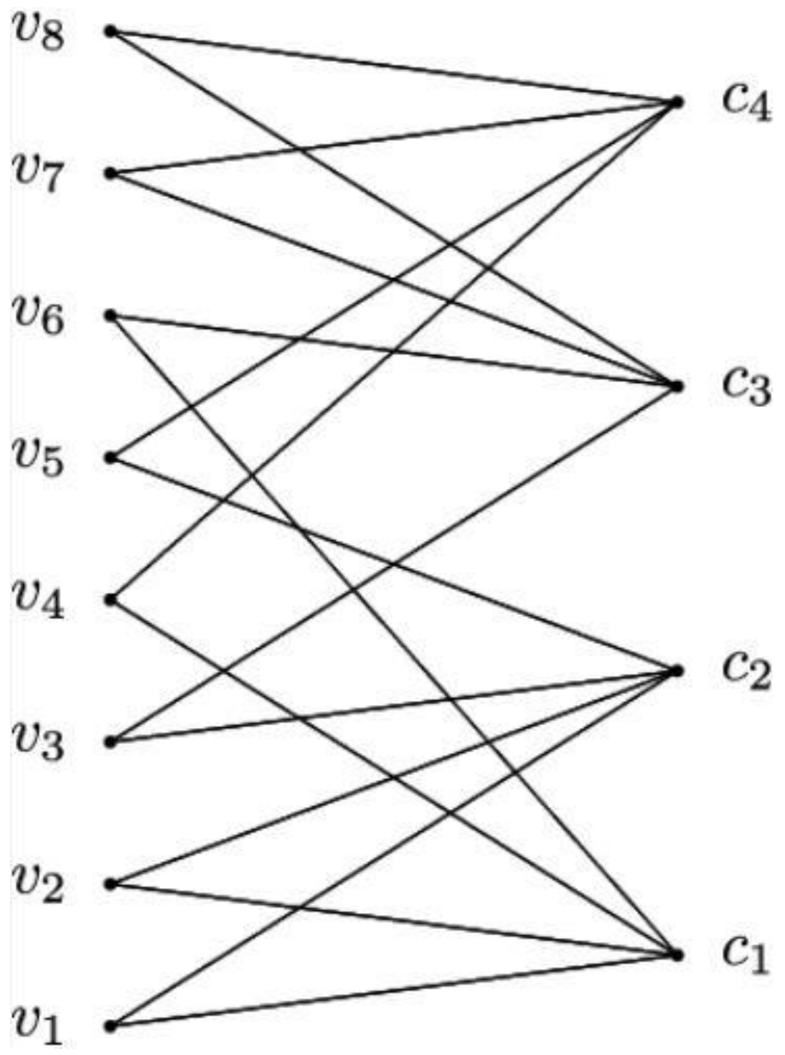
Wow, look at her



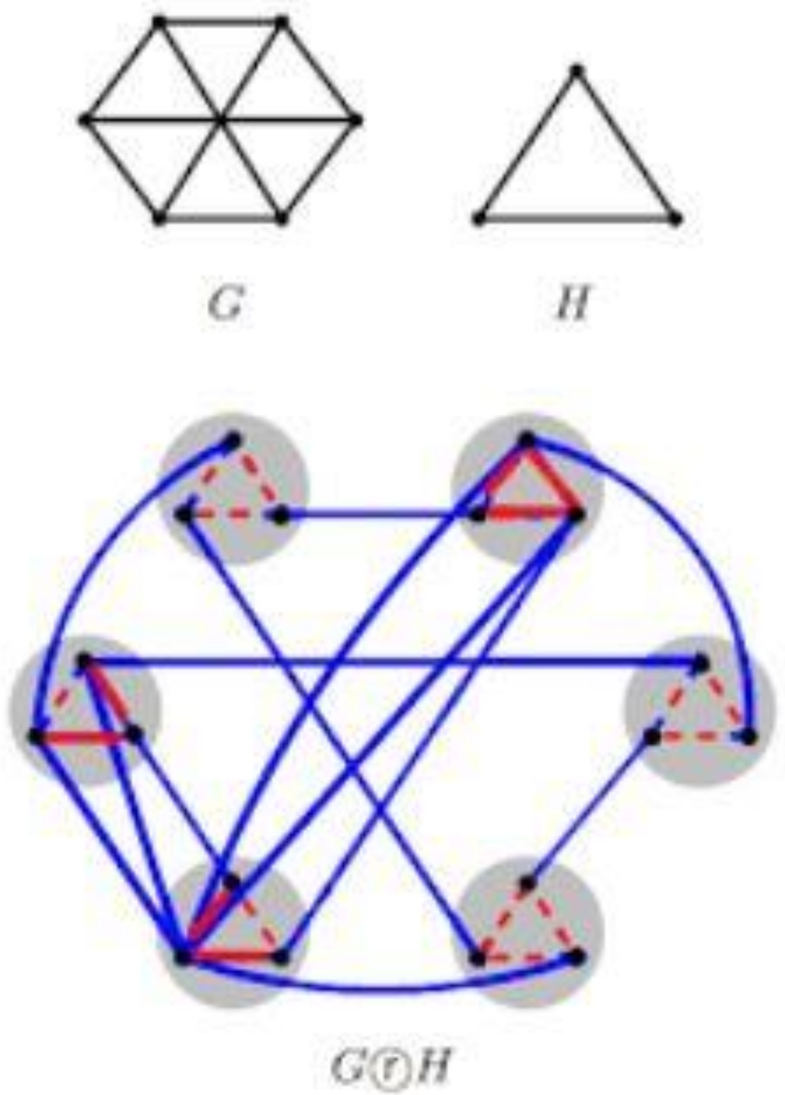
Stunning



Maybe she’s born with it



Maybe she was generated by a Zig-Zag Product



Introduction to Expanders

- There are a lot of definitions out there as to what constitutes a "good expander"
  - Certainly not a disconnected graph, though ☹
- Later, we want to focus on bipartite expanders with **good expansion**, i.e: every small subset of vertices in one partition of G has a satisfactorily large number of neighbors in the other.
- These pseudorandom graphs have nontrivial explicit constructions, the expansion factor of which varies between them…

Properties of Expanders

- We say that a d-regular graph G is a **good expander** if all of its adjacency matrix eigenvalues are small (spectral interpretation)
- Let $\Gamma(G)$ be the set of neighbors of a vertex $v \in V(G)$. For any set S of vertices, define the **boundary** of S as the cut $e(S, \bar{S})$.

The **edge-expansion** of G is defined by

$$h(G) = \min_{S:|S|\leq|V|} \frac{|\partial S|}{d \cdot |S|}$$

$h(G)$ is also known to be the **Cheeger constant**.

(Cheeger's Inequality): $\frac{1-\lambda_2}{2} \leq h(G) \leq \sqrt{2(1-\lambda_2)}$, where λ_2 is the second eigenvalue of the adjacency matrix of G.

(fun fact…computation of $h(G)$ is also NP-hard, though that's not the problem we're trying to solve.)

Some Brief Examples

- $h(G)$ is defined to be the smallest possible ratio between the size of a subgraph and the size of its boundary.
- A disconnected graph cannot be considered an expander; since $|E(S, \overline{S})| = 0$, it follows that $h(G) = 0$.
- Let $G = K_n$, or the complete graph on n vertices.

  Then $\left|E(S, \overline{S})\right| = |S| \cdot (n - |S|)$ for any subset $S \subseteq V(G)$. Since $G = K_n$,

$$h(G) = \min_{S:|S| \leq \frac{|V|}{2}} \frac{|\partial S|}{d \cdot |S|} = \min_{S:|S| \leq \frac{|V|}{2}} \frac{\left||S| \cdot (n - |S|)\right|}{d \cdot |S|} \approx \frac{1}{2}$$

Constructing Expanders: A Few Methods

- We can define an **expander family** to be a sequence of d-regular graphs $\{G_i\}_{i\in\mathbb{N}}$ with size that increases with i such that there is a constant $\epsilon > 0$ such that $h(G_i) \geq \epsilon$ for all i.
- **When discussing expander families, determining existence (and explicit construction) remain nontrivial problems!!**
- We have seen that complete graphs are notoriously excellent expanders, though they are not terribly sparse
- Biregular (d, c)-graphs have a high probability of being good expanders
- Others can be **combinatorically** or **probabilistically** generated

Bipartite Expanders

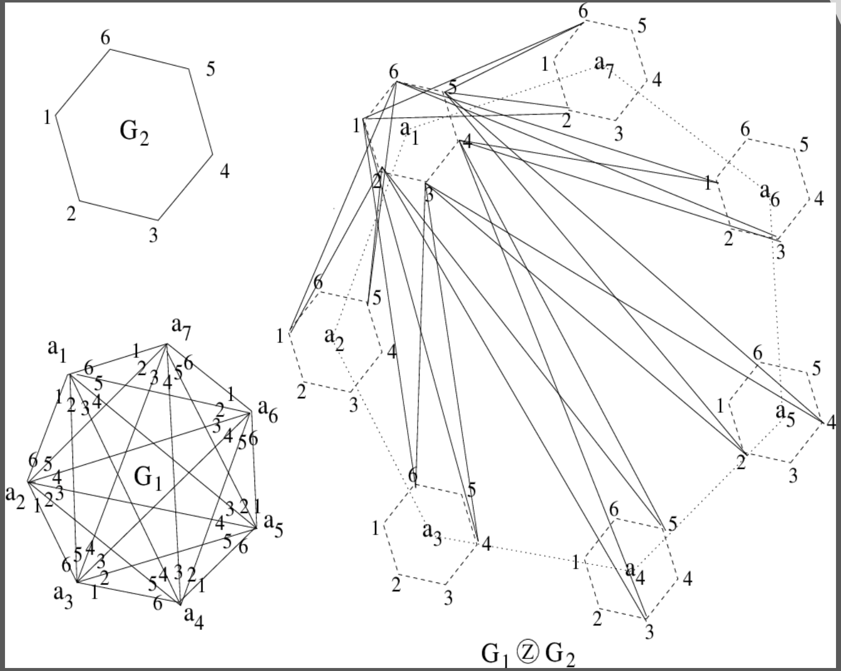
- Let $\mathcal{G}_{d,N}$ be a family of bipartite graphs with partite sets L, R of size N, such that $|L| = d$. For any d, there exists an $\alpha(d) > 0$, such that for all N it holds that

$$\boldsymbol{P}(G \text{ is a } (\alpha N, d-2) \text{ expander}) \geq \frac{1}{2}, \text{ where } G \text{ is chosen uniformly from } \mathcal{G}_{d,N}.$$

In other words, there is a relatively strong chance that, given an arbitrary bipartite graph, this graph will be a decent expander.

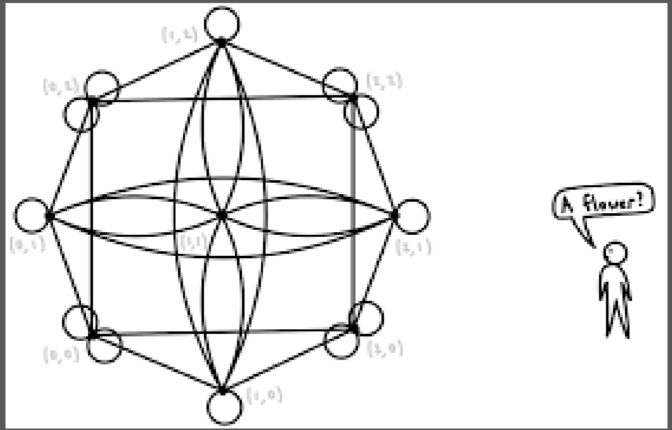
The Zig-Zag Product

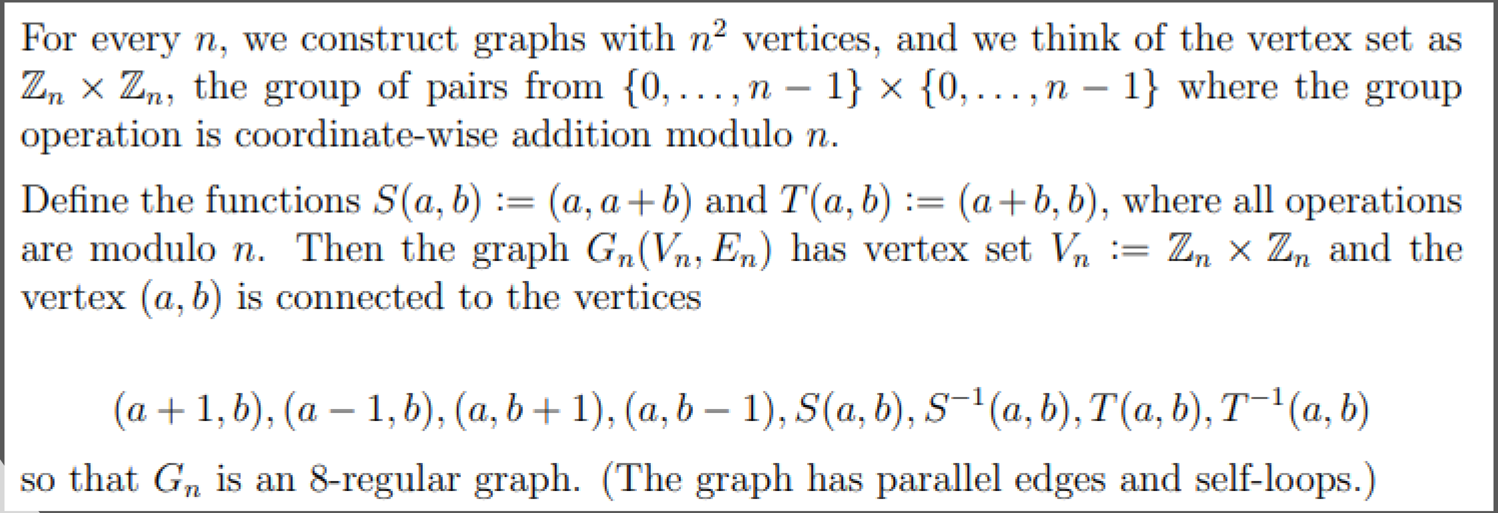
- Introduced by Reingold, Vadhan, and Widgerson in a 2002 article for *Annals of Mathematics*
- Bear with me here…
- Let X be a b –regular graph, and let e be an edge in X, and let v be a vertex incident to e. If e is a loop, then define $e(v) = v$; otherwise, define $e(v) = w$, where w is the other vertex incident to e. Else, $e(v)$ is undefined
- Let Y be a d –regular graph.
- Let E_X and E_Y be the edge-sets of X and Y.
- The **zig-zag product** $X \circledast Y$ is defined as follows:
  - The vertex set of $X \circledast Y$ is the Cartesian product $X \times Y$. If (x_1, y_1) and (x_2, y_2) are two vertices in this product, then the multiplicity of the edge between them is the number of ordered pairs $(z_1, z_2) \in E_X \times E_Y$ such that y_1 is an endpoint of z_1 and y_2 is an endpoint of z_2
- Summary: this lets us achieve the lower bound for the spectral gap mentioned for an expander adjacency matrix.



Margulis-Gabber-Galil

- First explicit construction, introduced in 1973!
- Given some $n \in \mathbb{N}$, construct an expander with n^2 vertices
- Formal definition and construction is given below:





For every n, we construct graphs with n^2 vertices, and we think of the vertex set as $\mathbb{Z}_n \times \mathbb{Z}_n$, the group of pairs from $\{0, \ldots, n-1\} \times \{0, \ldots, n-1\}$ where the group operation is coordinate-wise addition modulo n.

Define the functions $S(a,b) := (a, a+b)$ and $T(a,b) := (a+b, b)$, where all operations are modulo n. Then the graph $G_n(V_n, E_n)$ has vertex set $V_n := \mathbb{Z}_n \times \mathbb{Z}_n$ and the vertex (a,b) is connected to the vertices

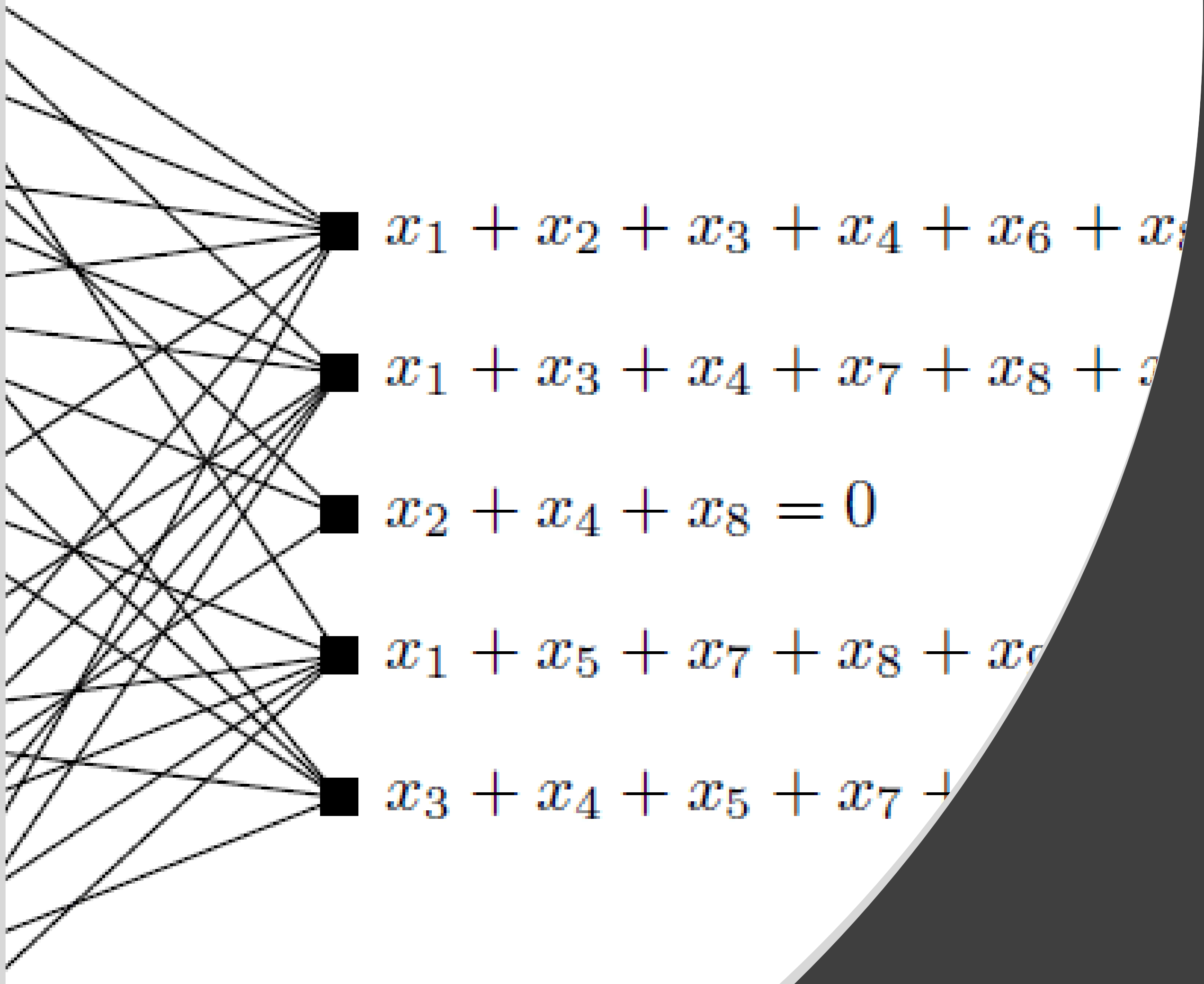
$$(a+1,b), (a-1,b), (a,b+1), (a,b-1), S(a,b), S^{-1}(a,b), T(a,b), T^{-1}(a,b)$$

so that G_n is an 8-regular graph. (The graph has parallel edges and self-loops.)

Information Theory Primer

Vocabulary

- **Rate:** A fractional number that expresses what part of the message is actually meaningful. The bigger the rate, the stronger the code.
- **Codeword:** A bitstring representation of a symbol.
- **Distance:** The number of bits which differ between a sent codeword and received codeword
- **Block Length:** The column length of the parity check matrix.
- **Parity Condition:** Constraints which correspond to the equations satisfied by the elements of the rows of a parity-check matrix H.


$x_1 + x_2 + x_3 + x_4 + x_6 + x$
$x_1 + x_3 + x_4 + x_7 + x_8 +$
$x_2 + x_4 + x_8 = 0$
$x_1 + x_5 + x_7 + x_8 + x$
$x_3 + x_4 + x_5 + x_7 +$

Introduction to Linear Codes

- Error-Correcting Codes
  - Claude Shannon: *A Mathematical Theory of Communication* (1948)
- Probabilistic (Shannon) versus Combinatorial (Hamming) viewpoint
- Left: LPDC code (a generalization of the expander code!! More on that later.)
- A **linear code** over a field $\mathbb{Z}_q$ is a subspace of $\mathbb{Z}_q^n$
- Here, we'll refer to codes over the field $\mathbb{Z}_2$ (integers modulo 2)
  - Also known as $\{0,1\}$.

The Goal

$$\mathbf{G}^{\mathrm{T}} := \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \qquad \mathbf{H} := \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

- The Hamming (7,4) code was invented in 1950 by mathematician Richard Hamming when he became frustrated with a faulty punch card reader while working for Bell Labs
- The intention was to produce a code that would not only detect flipped transmission bits, but also correct errors, once found.
- That way, the integrity of a communications channel can be made more stable, even in the presence of noise.

Parity Check Matrices & Generator Matrices

$dH^T = \text{res}$

$$\mathbf{G}^\mathrm{T} := \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \qquad \mathbf{H} := \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

- Consider an $[n, m]_2$ linear code
- $H = [-P^T \,|I_m]$ (standard form)
  - $(n-m) \times n$ matrix
  - If our field is $GF(2)$, or $\mathbb{Z}_2$, then $-P^T = P^T$, i.e., the negative doesn't matter.
  - Rows are independent
  - If $\boldsymbol{c}$ is a codeword, then $\boldsymbol{c}H^T = 0$, and similarly in the other direction
- $G = [I_{n-m}|P]$ (standard form)
  - $m \times n$ matrix
- $GH^T = P - P = 0$
- Hence, we can generate G from H and vice versa

Quick Example (Generation, Conversion, Encoding, Decoding)

- Let $H = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix}$. Then $-P^T = P^T = \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$
  - This is a $(n-m) \times n = 3 \times 5$ matrix. So the identity for the block matrix G is $I_m = I_2$, and $P = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$.

Then $G = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$

- The rows of H represent the coefficients of the parity check equations $c_1 + c_2 + c_3 = 0, c_2 + c_4 = 0$ and $c_1 + c_5 = 0$.

Quick Example (Generation, Conversion, Encoding, Decoding)

- Let $M = [1 \quad 1]$, with H and G defined previously. Let C be the set of possible codewords.

Then $\boldsymbol{c} = MG = [1 \quad 1] \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} =$ $\boxed{[1 \ 1 \ 0 \ 1 \ 1]}$

is a codeword produced by G. (Remember, computations here are performed mod 2)

- H is then used to determine whether a received (potentially-degraded) message M' is a codeword of C.

Quick Example (Generation, Conversion, Encoding, Decoding)

- Let $M = [1 \quad 1]$, with H and G defined previously. Let C be the set of possible codewords.

Then $\boldsymbol{c} = MG = [1 \quad 1]\begin{bmatrix}1 & 0 & 1 & 0 & 1\\ 0 & 1 & 1 & 1 & 0\end{bmatrix} =$ $[1 \ 1 \ 0 \ 1 \ 1]$

is a codeword produced by G. (Remember, computations here are performed mod 2)

- H is then used to determine whether a received (potentially-degraded) message M' is a codeword of C.
- Define the **syndrome** $s(M')$ to be the result of $M'H^T$.

Quick Example (Generation, Conversion, Encoding, Decoding)

- Let $M = [1 \quad 1]$, with H and G defined previously. Let C be the set of possible codewords.

Then $\boldsymbol{c} = MG = [1 \quad 1]\begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} = [1\ 1\ 0\ 1\ 1]$

is a codeword produced by G. (Remember, computations here are performed mod 2)

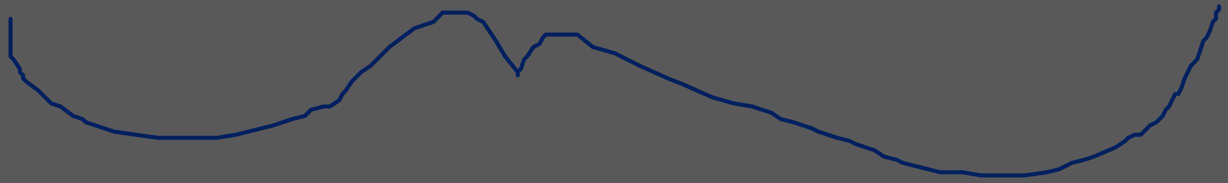
- H is then used to determine whether a received (potentially-degraded) message M' is a codeword of C.
- Define the **syndrome** $s(M')$ to be the result of $M'H^T$.
  - If the transmitted word is the same as the original codeword, then $M'H^T = \mathbf{0}$.
  - If the code has errors, $s(M')$ can be used to determine for what bits an error has occurred.

$c h_1 + c_2 h \ldots = 0$

$d \longrightarrow \longrightarrow c$ $\quad d + e = (c)$

Decoding

- The original codeword $\boldsymbol{c}$ must be the closest codeword to d relative to distance: $d(\boldsymbol{c}, d) \leq d(\boldsymbol{c}', d)$ for all other codewords $\boldsymbol{c}'$.
- We need to find the smallest number of digits that differ between $\boldsymbol{c}$ and d.
- **Solution:** iterate through the set of possible error-words and find the word e with the smallest weight
  - **Then $\boldsymbol{c} = d - e$.**
- **Unfortunately, this can be quite inefficient…**



Decoding

- This technique forces us to store the entire space $\mathbb{Z}_q^n$, search for d, and then also for the coset $d + C$.
  - Depending on the size of n, things can become pretty big.

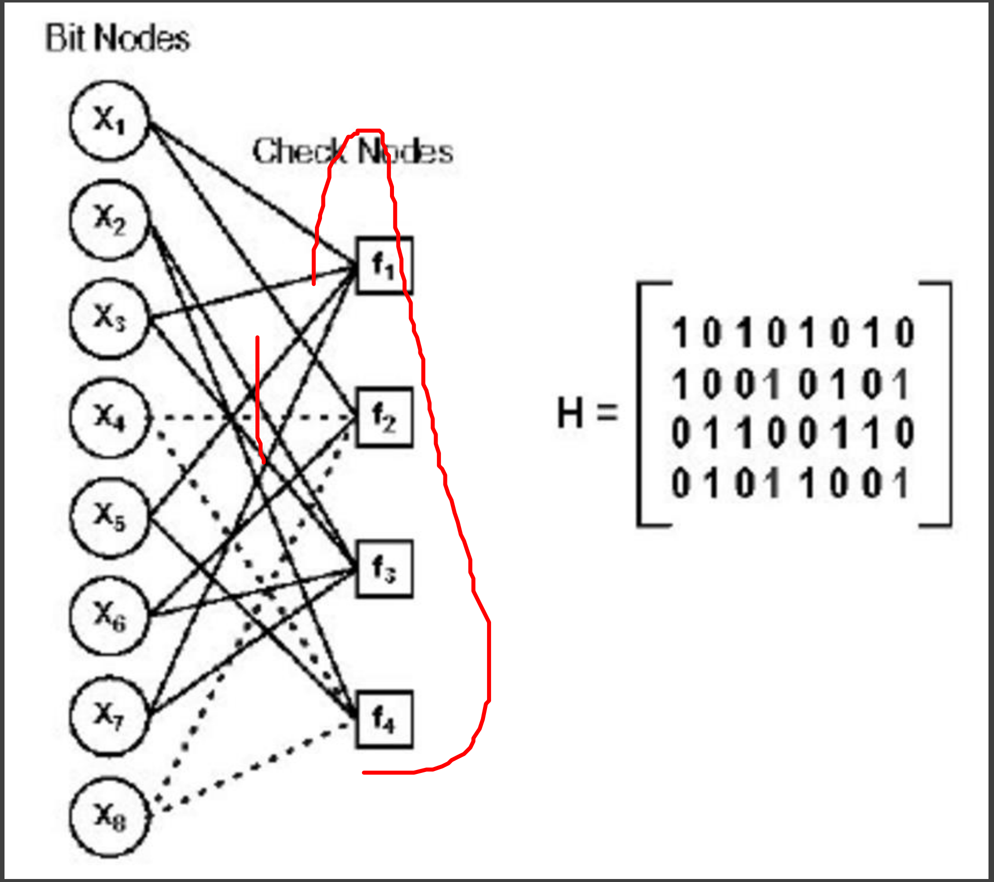
We Could REALLY Benefit From Optimization Here…

How to Tell if a Code is Good?

- ✓ Fast (polynomial-time) decoding
- ✓ CORRECT decoding (good rate)
- ✓ Good distance

- ✓ Expander codes have:
  - ✓ Constant positive rate
  - ✓ Constant positive relative distance
  - ✓ Constant alphabet size
  - ✓ Can be encoded and decoded in time <u>proportional to the block length of the code</u>

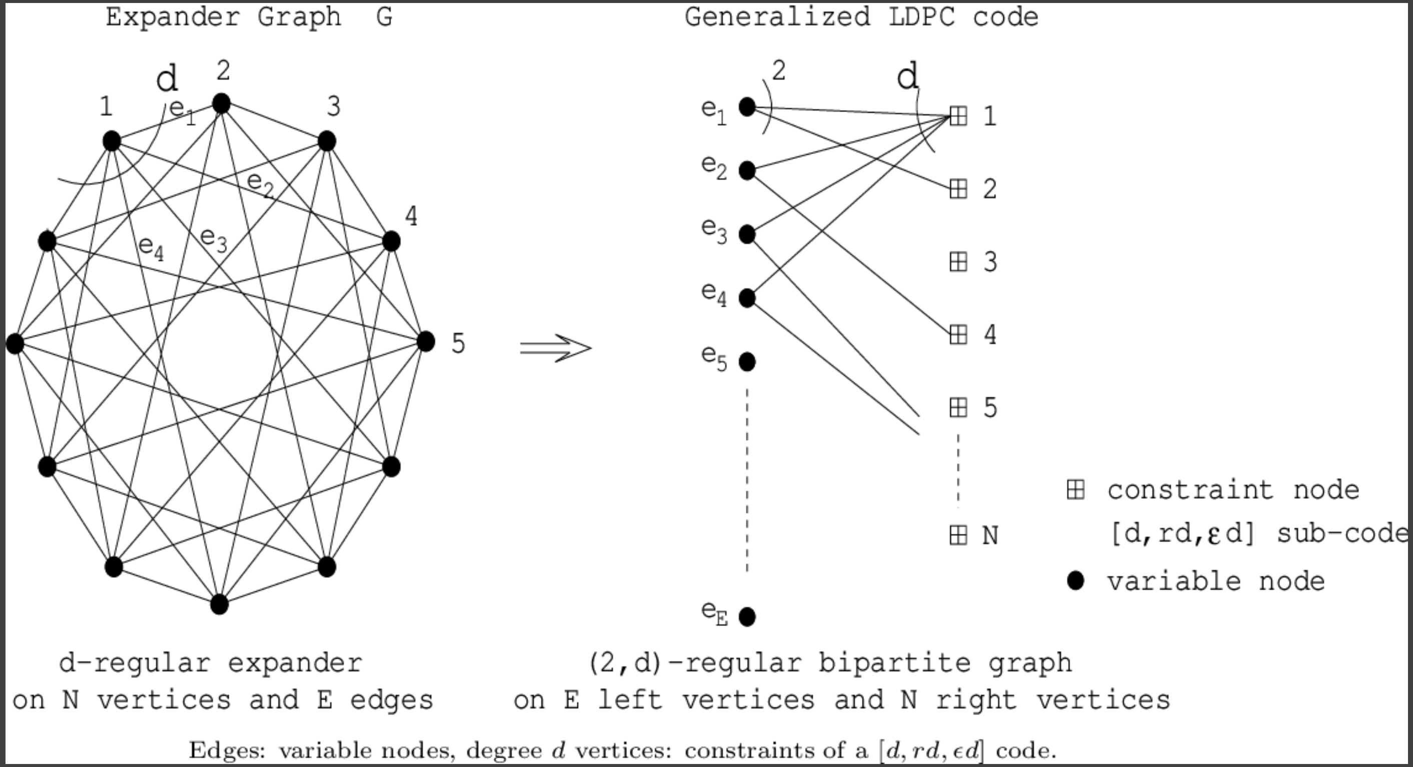
Relationship Between Expanders & Parity Check Matrices



$$H = \begin{bmatrix} 1&0&1&0&1&0&1&0 \\ 1&0&0&1&0&1&0&1 \\ 0&1&1&0&0&1&1&0 \\ 0&1&0&1&1&0&0&1 \end{bmatrix}$$

- Define the *factor graph* F of a linear $[n, n-m]_2$ code G to be the bipartite graph representative of the parity-check matrix H. Let F be organized so that
  - There is a vertex in the left partition of F corresponding to each bit of a codeword in H
  - There is a vertex in the right partition of F corresponding to each parity check
  - An edge (u, v) exists in the left and right partitions iff the bit mapped to u is involved in the parity check corresponding to v
- This construction establishes a correspondence between bipartite graphs and linear codes.

Expander Codes



Edges: variable nodes, degree d vertices: constraints of a $[d, rd, \epsilon d]$ code.

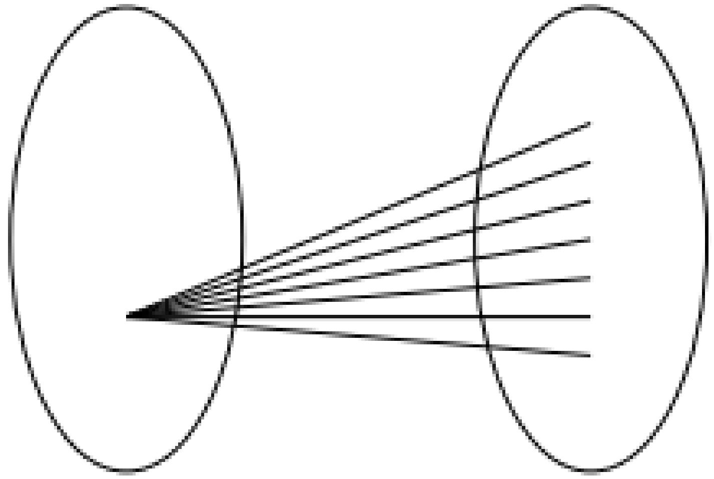
- Formally, an **expander code** is a linear block code $[n, n-m]_2$ whose factor graph - i.e., parity check matrix H - is produced by the adjacency matrix of a bipartite expander graph.
- The linear growth rate of the expander allows for efficient (read: polynomial-time) encoding and decoding schemes
  - Encoding is usually around $\boxed{O(n^2)}$, and decoding has been shown to be as fast as $O(n)$.
  - It has been shown by Spielman that we can also perform $\underline{O(n) \text{ encodings}}$
- LDPC (low-density parity check) codes are similar generalizations of these to sparse bipartite graphs

Philosophy

- Let C be the set of strings which satisfy every parity check using H. Each codeword can be viewed as a set of edges in a graph G, that induce codewords of C at each vertex.
- There is a family of expander codes $\mathcal{C}(\mathcal{G}, C)$ for which there is a linear-time algorithm that will map to a codeword any word of relative distance at most ϵ from that codeword.
- If a vertex is flipped, then, due to the connectivity of this vertex, all the constraints on that vertex will be violated. If a vertex finds itself to be involved every time **a large number of constraints** is violated, majority-rule dictates that it should be flipped.

Hence, the better the expansion, the better the code.

[illegible] $\longrightarrow \epsilon$



Encoding

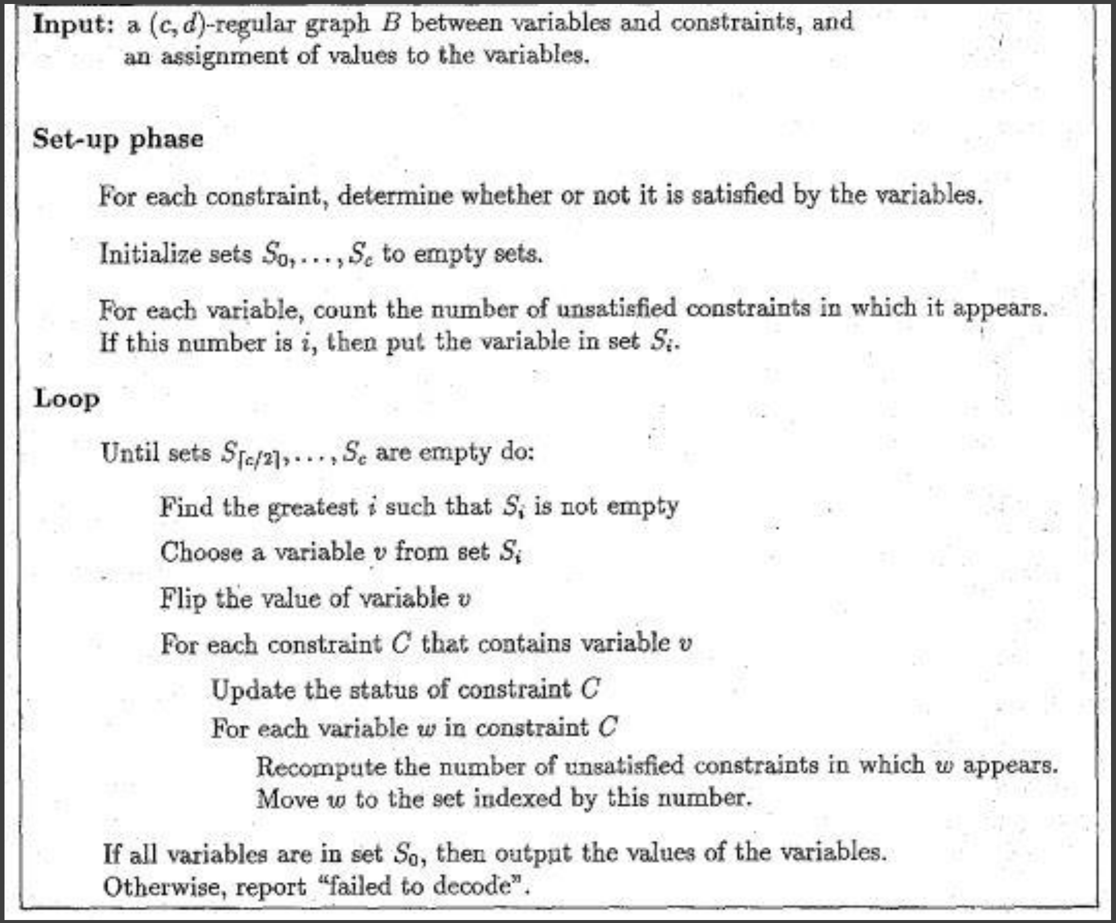
- Recall the formulation of the bipartite expander B from the beginning.
- Specifically, label the nodes in the larger partition as **variables**, with $|L| = n$. The nodes in the smaller partition are called **constraints**, with $|S| = \frac{cn}{d}$.
- The parity check matrix corresponds precisely with the adjacency matrix of B; edges between vertices represent codewords!

Decoding

- Let $M = x_1 x_2 \dots x_N$. Let L be the partite set consisting of variable vertices, and let R be the partite set consisting of constraint vertices
- A vertex $v \in L$ is **satisfied** if and only if the **parity condition** involving v is satisfied for M. For each $i \in R$, count the number of satisfied and unsatisfied neighbors of i. If i has more unsatisfied than satisfied neighbors, we have enough suspicion that we need to flip x_i.
- Repeat until each v is satisfied, which is to say that M becomes a codeword.
- **This algorithm should take $O(n)$ iterations.**

Spielman Decoding

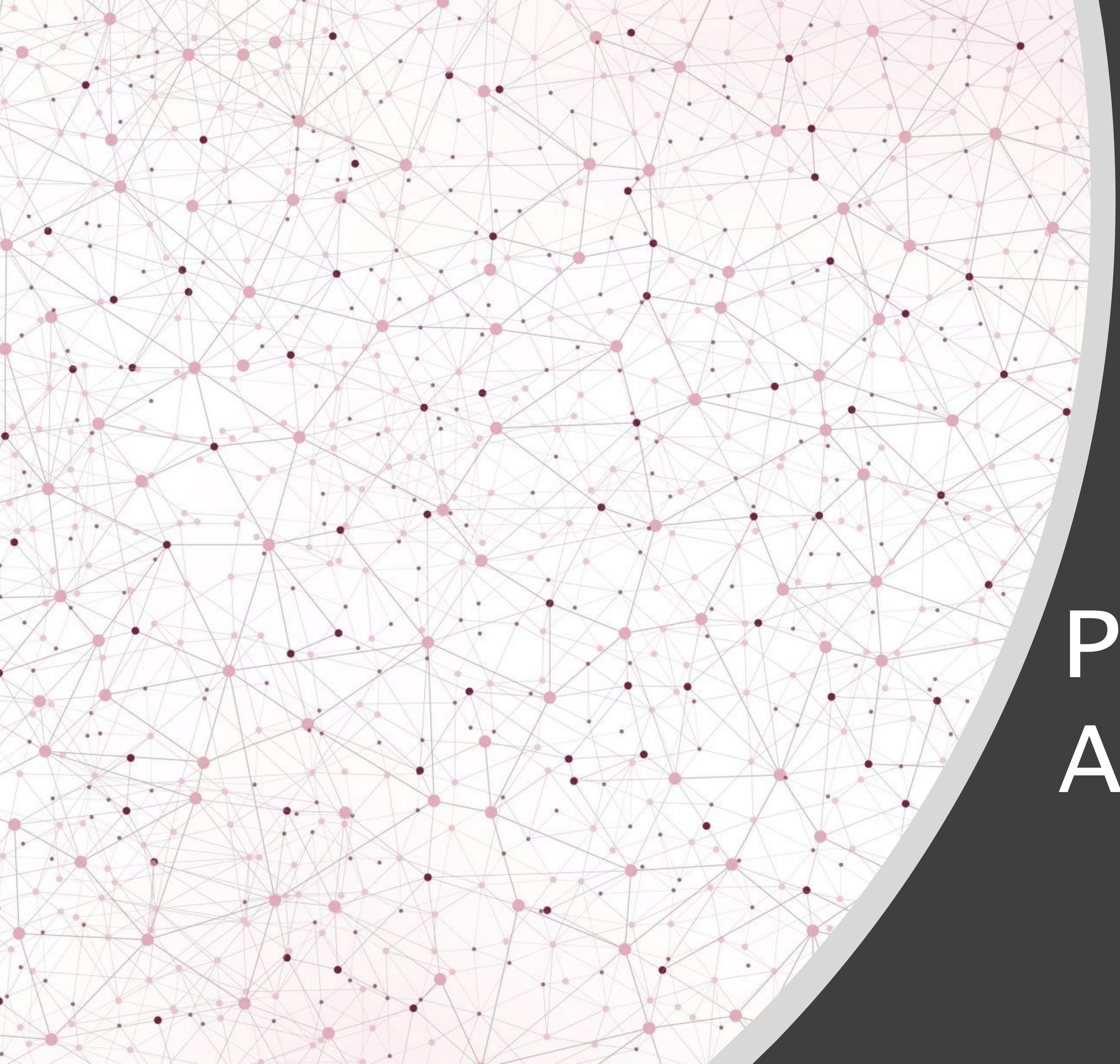
- Strategy: flip an edge if any neighbors to an edge thinks the edge should be flipped.
- The following algorithm (right) is due to Spielman:



```
Input: a $(c, d)$-regular graph $B$ between variables and constraints, and
       an assignment of values to the variables.

Set-up phase

    For each constraint, determine whether or not it is satisfied by the variables.

    Initialize sets $S_0, \ldots, S_c$ to empty sets.

    For each variable, count the number of unsatisfied constraints in which it appears.
    If this number is $i$, then put the variable in set $S_i$.

Loop

    Until sets $S_{\lceil c/2 \rceil}, \ldots, S_c$ are empty do:
        Find the greatest $i$ such that $S_i$ is not empty
        Choose a variable $v$ from set $S_i$
        Flip the value of variable $v$
        For each constraint $C$ that contains variable $v$
            Update the status of constraint $C$
            For each variable $w$ in constraint $C$
                Recompute the number of unsatisfied constraints in which $w$ appears.
                Move $w$ to the set indexed by this number.

    If all variables are in set $S_0$, then output the values of the variables.
    Otherwise, report "failed to decode".
```

How to Tell if a Code is Good?

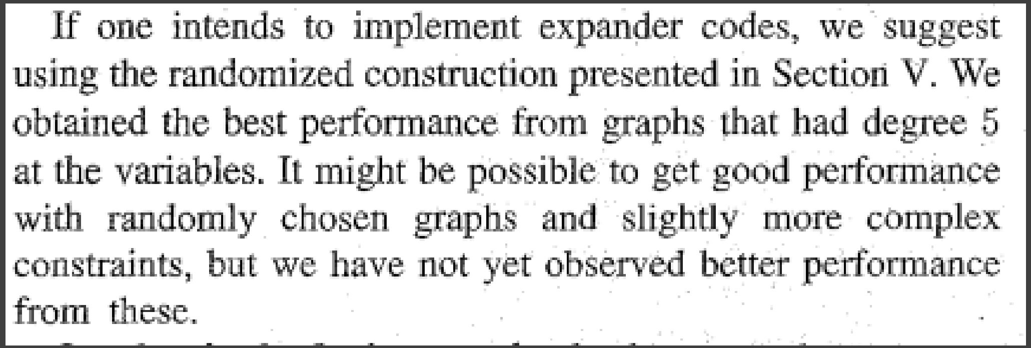
- If the expander we use is a good expander, then the resulting code will be a good code.
- (!) The difficulty here is that we assume the expander in question has **good expansion**. Generating good expanders randomly is a bit of a toss-up, but makes more practical sense than the explicit construction.



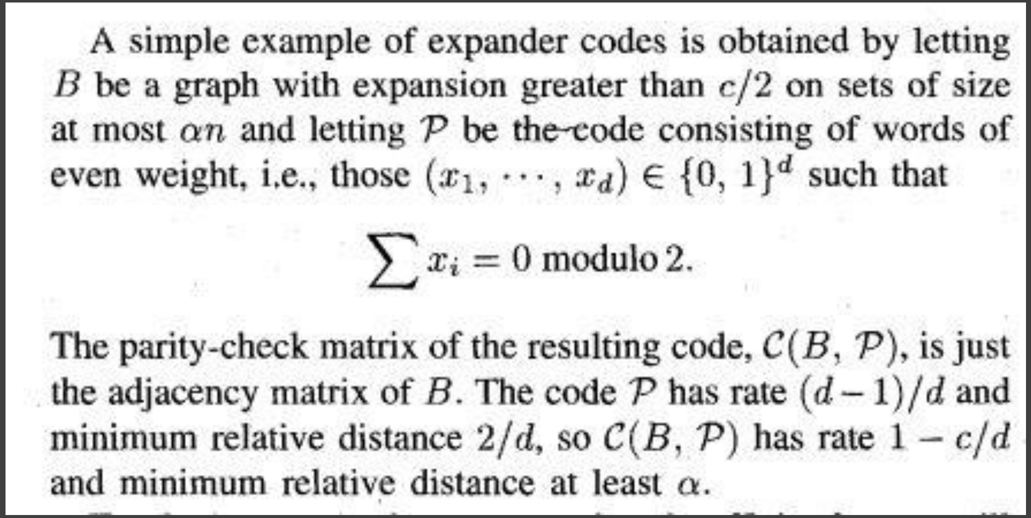
Part II: Application

What'll Be Discussed

- Creating a suitable bipartite expander
- Calculating its parity matrix and generator matrix
- Correct?



If one intends to implement expander codes, we suggest using the randomized construction presented in Section V. We obtained the best performance from graphs that had degree 5 at the variables. It might be possible to get good performance with randomly chosen graphs and slightly more complex constraints, but we have not yet observed better performance from these.



A simple example of expander codes is obtained by letting B be a graph with expansion greater than $c/2$ on sets of size at most αn and letting $\mathcal{P}$ be the code consisting of words of even weight, i.e., those $(x_1, \cdots, x_d) \in \{0, 1\}^d$ such that

$$\sum x_i = 0 \text{ modulo } 2.$$

The parity-check matrix of the resulting code, $\mathcal{C}(B, \mathcal{P})$, is just the adjacency matrix of B. The code $\mathcal{P}$ has rate $(d-1)/d$ and minimum relative distance $2/d$, so $\mathcal{C}(B, \mathcal{P})$ has rate $1 - c/d$ and minimum relative distance at least α.

Creating a Suitable Expander Code

- We have many options here; simplest is to use a random biregular (d,c) graph, which has high probability of being a good expander. We'll be using the suggestions by Spielman (left).
- We'll do a by-hand example first with an ordinary bipartite graph (which is also an expander, but might not have the expansion properties we want…), and then demonstrate some code for generating these codes
- The partite sets have orders n and $\frac{cn}{d}$.
- Additionally, we should use the Cheeger Inequalities to make sure that our calculated $h(G)$ value makes sense. There's code for that too.

Creating a Suitable Expander Code

Let B be the graph on the right.

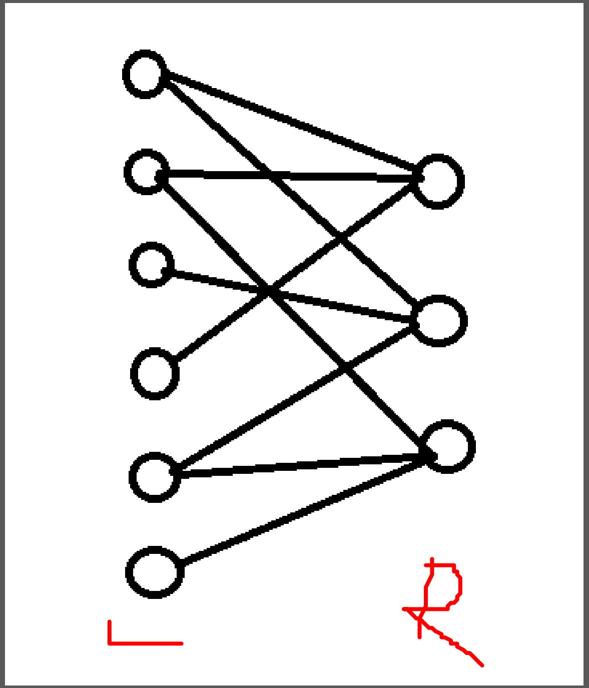
Here is its adjacency matrix:

$$A = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \end{bmatrix}$$

After converting A to standard form, we get the parity-check matrix

$$H = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

And generator matrix $G = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$



One Round of Decoding By Hand

Let $M = [1\ \ 0\ \ 1]$. Then $MG = [1\ \ 0\ \ 1\ \ 1\ \ 0\ \ 0]$ is a codeword in C. Let M' be the received message, with $M' = [1\ \ 1\ \ 1\ \ 1\ \ 0\ \ 0]$ (so we differ only by one bit, for simplicity.)

Recall H. The vertex v is satisfied if the parity condition involving v is satisfied by M':

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

	v_1	v_2	v_3	v_4	v_5	v_6
$v_7[0]$	1	1	0	1	0	0
$v_8[1]$	1	0	1	0	1	0
$v_9[2]$	1	1	1	0	0	1

Comparing with the message, we find that the variables v_1, v_2, and v_4 are unsatisfied relative to v_7.

1 1 1 1 0 0

✓ ✓ ✓ ✓

$c_1 + c_2 + c_4 \to 0$

$= 1\,1\,1 \neq 0$

One Round of Decoding By Hand

Let $M = [1 \;\; 0 \;\; 1]$. Then $MG = [1 \;\; 0 \;\; 1 \;\; 1 \;\; 0 \;\; 0]$ is a codeword in C. Let M' be the received message, with $M' = [1 \;\; 1 \;\; 1 \;\; 1 \;\; 0 \;\; 0]$ (so we differ only by one bit, for simplicity.)

Recall H. The vertex v is satisfied if the parity condition involving v is satisfied by M':

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

	v_1	v_2	v_3	v_4	v_5	v_6
$v_7[0]$	1	1	$\cancel{0}$	1	$\cancel{0}$	0
$v_8[1]$	1	0	1	0	1	0
$v_9[2]$	1	1	1	0	0	1

Comparing with the message, we find that the variables v_1, v_2, and v_4 are unsatisfied relative to v_7. We **don't** flip the first bit of M. (If we had higher connectivity, i.e., fewer zero entries, we would have been able to find out a lot faster…)

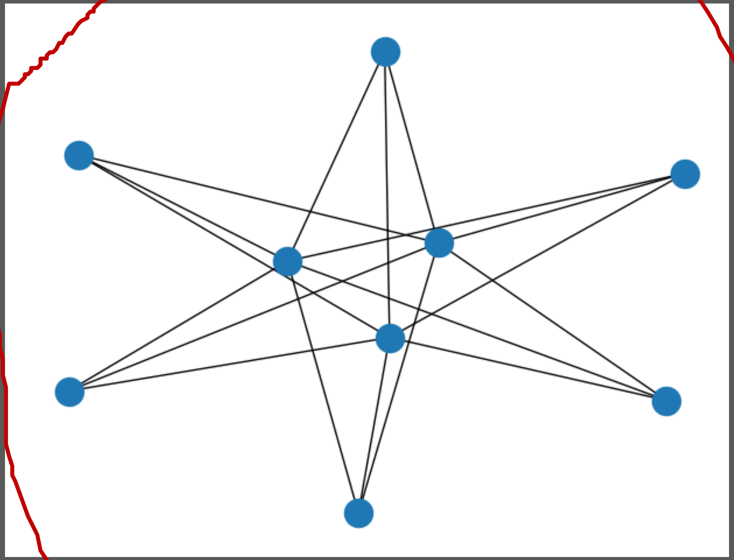
Rudimentary Encoding & Graph Generation Simulation in Python

- Generate a bipartite expander with **nx.algorithms.bipartite.random_graph** and get its adjacency matrix
  - Compute the second eigenvalue of the adjacency matrix for $h(G)$ evaluation
  - Alternatively, we can also compute the union of two Margulis constructions and then trim edges accordingly to achieve better expansion
- Use the adjacency matrix to compute G
  - Runtime: $O(n^2)*$
- Validate that the rate and correctness correspond to the theorized values
- Efficient decoding is very difficult to implement in-practice

d c

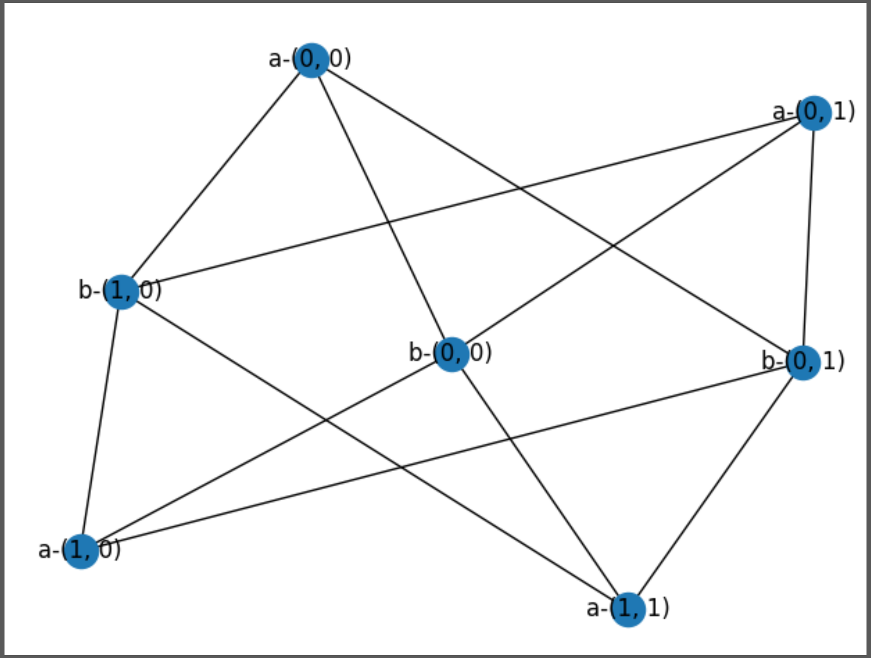
Expander Generation

- First way (naiive)
- A bit hacky, but we start with a complete bipartite graph
- Trim the edges until it is satisfactorily d,c regular (if possible)
- If not possible, return failure and try different parameters

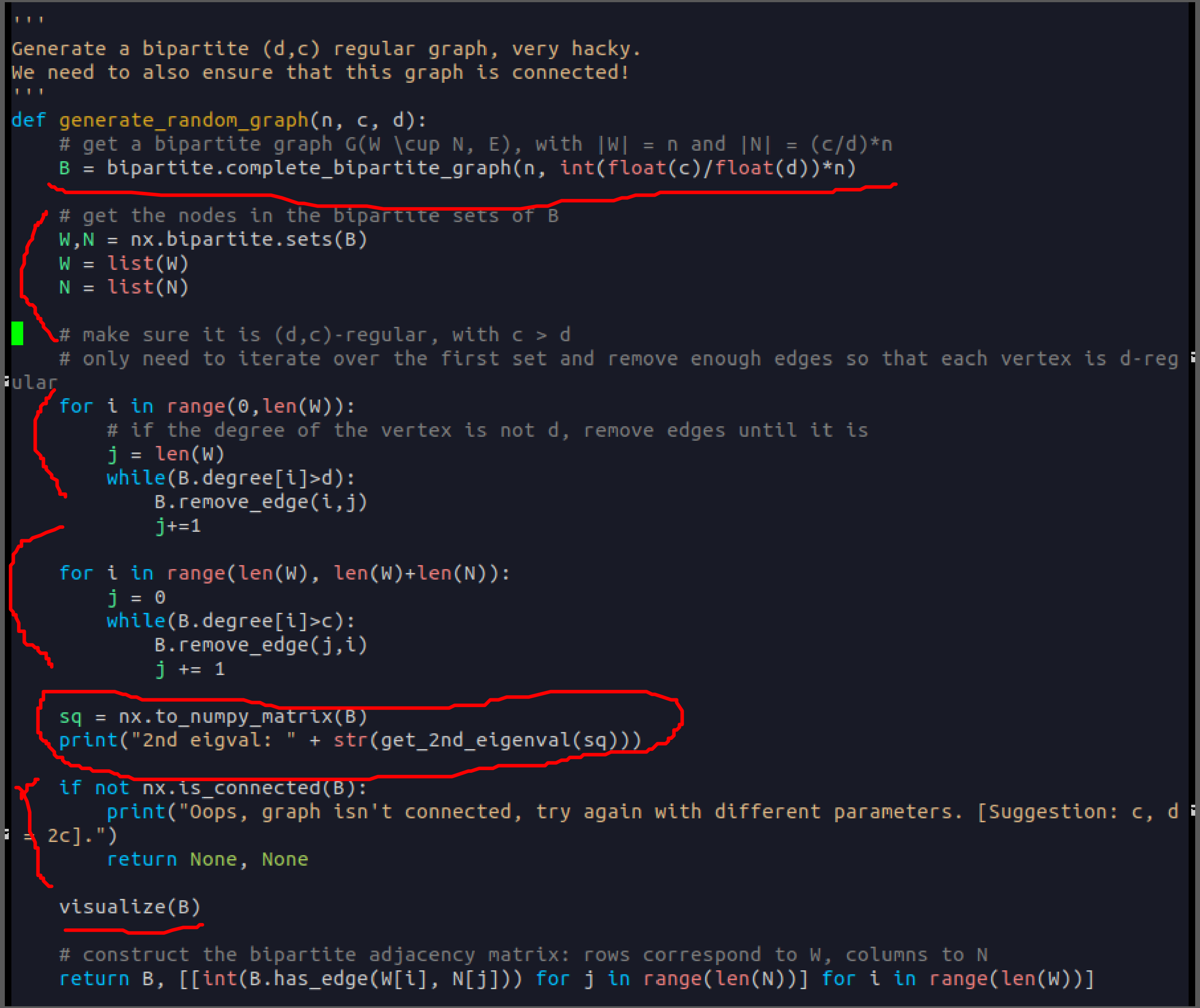


Expander Generation

- Second (more legitimate) way:
- Generate an ordinary d-regular expander
- Generate two copies of $V(G)$, called V_1 and V_2.
- An edge v_1v_2 exists in G' only when v_1v_2 exists in G for $v_1 \in V_1$ and $v_2 \in V_2$.
- The expansion of G' follows from the expansion of G.

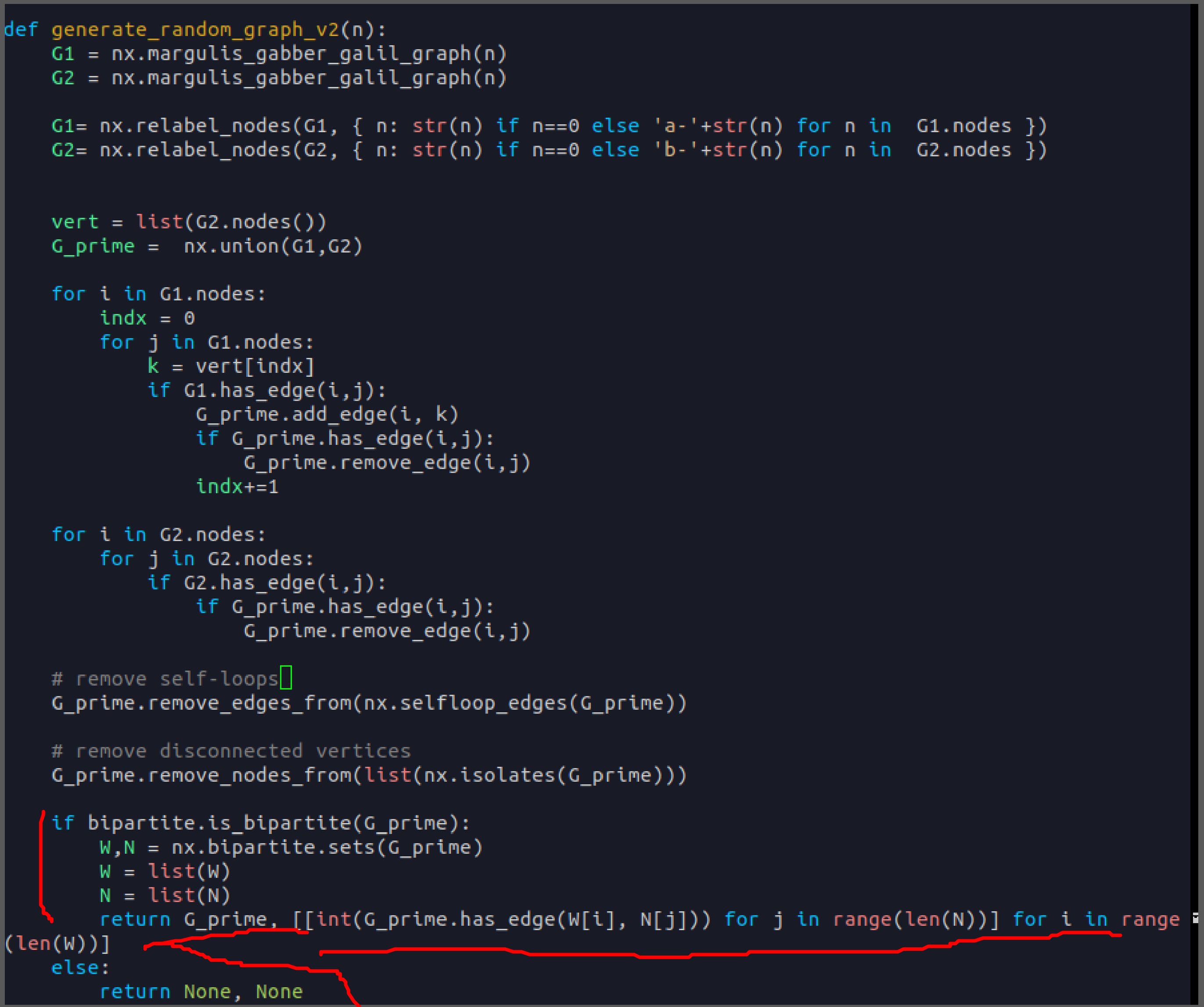


Expander Generation (v1)



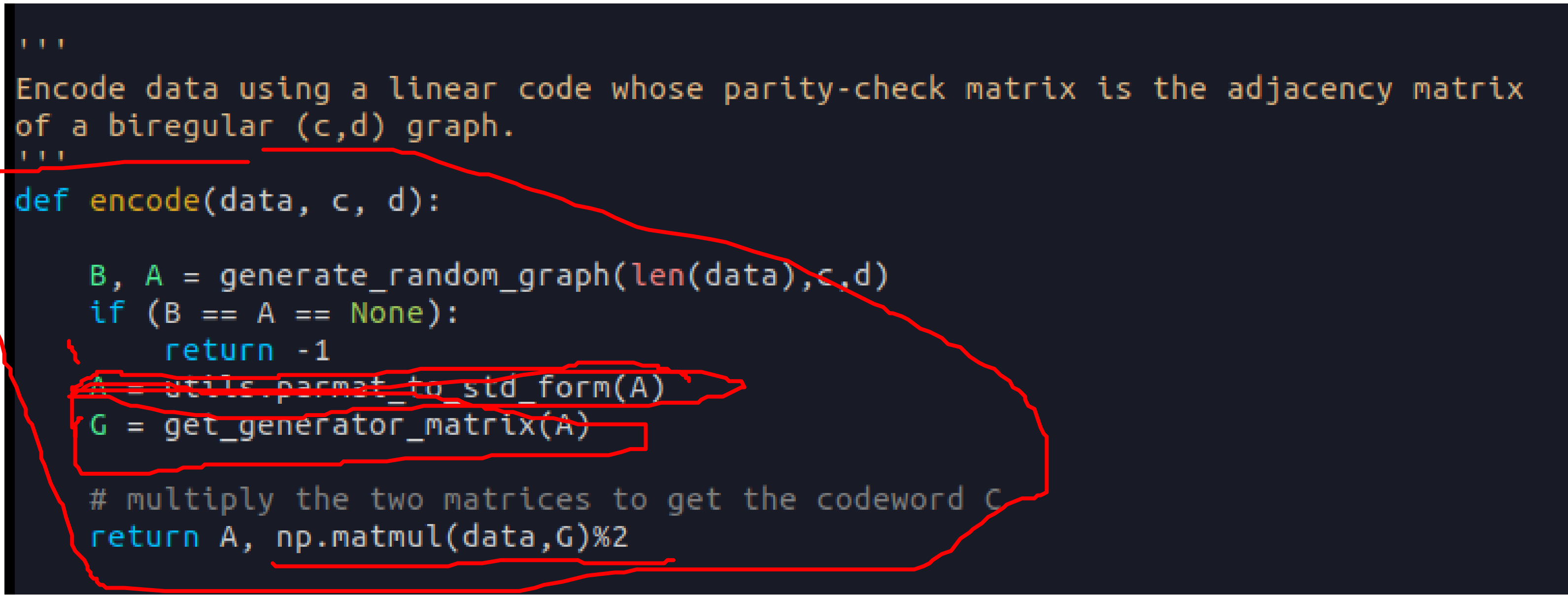
```
'''
Generate a bipartite (d,c) regular graph, very hacky.
We need to also ensure that this graph is connected!
'''
def generate_random_graph(n, c, d):
    # get a bipartite graph G(W \cup N, E), with |W| = n and |N| = (c/d)*n
    B = bipartite.complete_bipartite_graph(n, int(float(c)/float(d))*n)

    # get the nodes in the bipartite sets of B
    W,N = nx.bipartite.sets(B)
    W = list(W)
    N = list(N)

    # make sure it is (d,c)-regular, with c > d
    # only need to iterate over the first set and remove enough edges so that each vertex is d-reg
ular
    for i in range(0,len(W)):
        # if the degree of the vertex is not d, remove edges until it is
        j = len(W)
        while(B.degree[i]>d):
            B.remove_edge(i,j)
            j+=1

    for i in range(len(W), len(W)+len(N)):
        j = 0
        while(B.degree[i]>c):
            B.remove_edge(j,i)
            j += 1

    sq = nx.to_numpy_matrix(B)
    print("2nd eigval: " + str(get_2nd_eigenval(sq)))

    if not nx.is_connected(B):
        print("Oops, graph isn't connected, try again with different parameters. [Suggestion: c, d
 = 2c].")
        return None, None

    visualize(B)

    # construct the bipartite adjacency matrix: rows correspond to W, columns to N
    return B, [[int(B.has_edge(W[i], N[j])) for j in range(len(N))] for i in range(len(W))]
```

Expander Generation (v2)

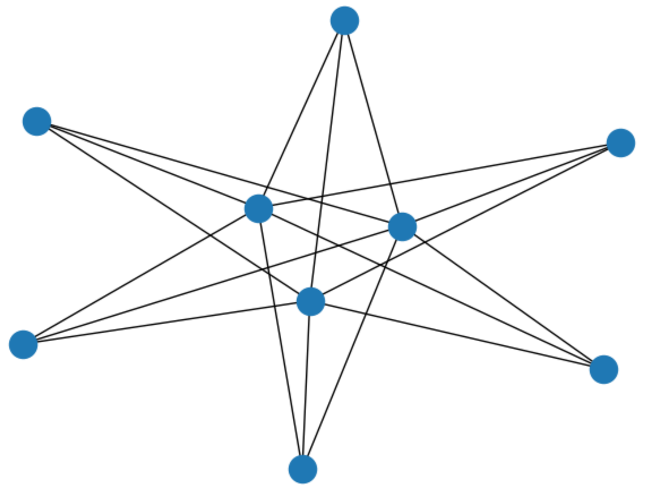


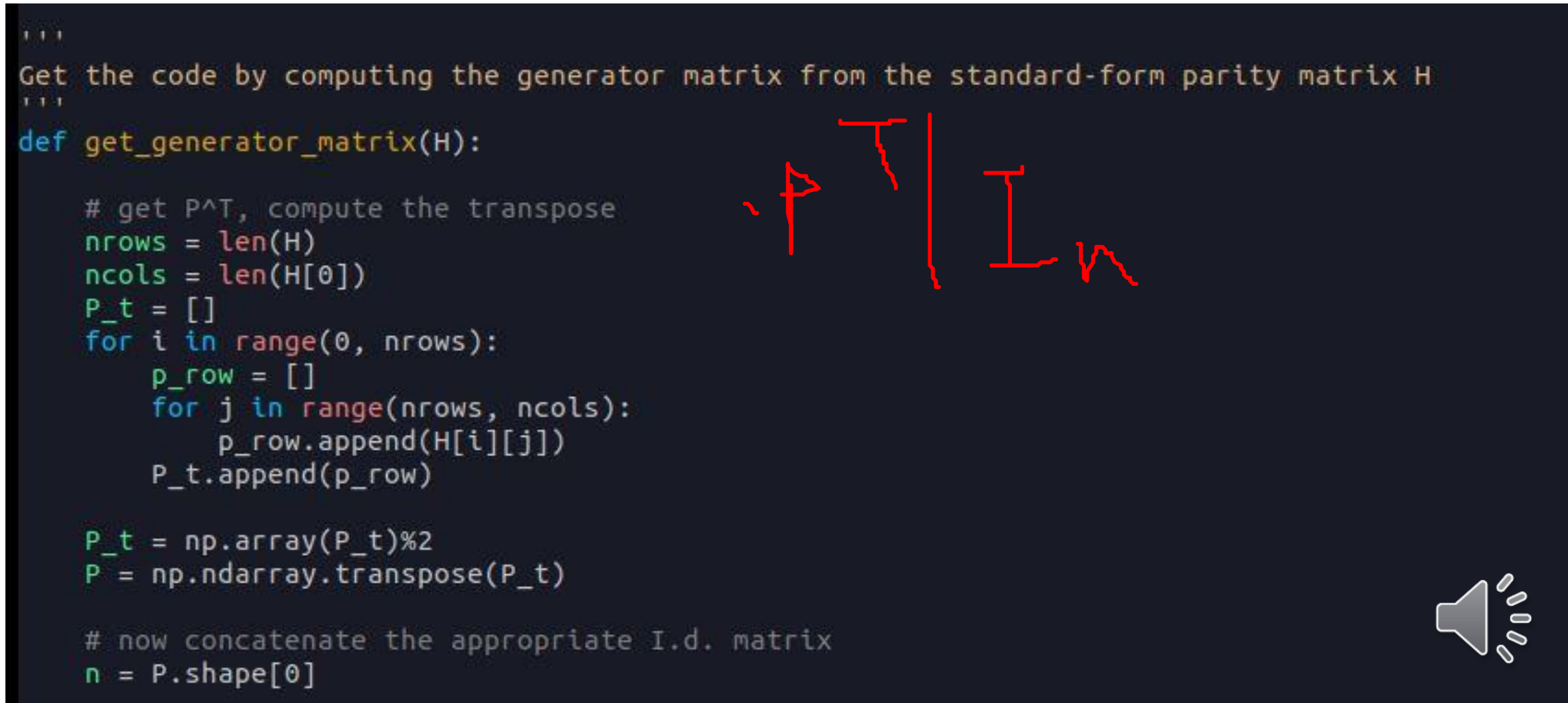
```
def generate_random_graph_v2(n):
    G1 = nx.margulis_gabber_galil_graph(n)
    G2 = nx.margulis_gabber_galil_graph(n)

    G1= nx.relabel_nodes(G1, { n: str(n) if n==0 else 'a-'+str(n) for n in  G1.nodes })
    G2= nx.relabel_nodes(G2, { n: str(n) if n==0 else 'b-'+str(n) for n in  G2.nodes })


    vert = list(G2.nodes())
    G_prime =  nx.union(G1,G2)

    for i in G1.nodes:
        indx = 0
        for j in G1.nodes:
            k = vert[indx]
            if G1.has_edge(i,j):
                G_prime.add_edge(i, k)
                if G_prime.has_edge(i,j):
                    G_prime.remove_edge(i,j)
                indx+=1

    for i in G2.nodes:
        for j in G2.nodes:
            if G2.has_edge(i,j):
                if G_prime.has_edge(i,j):
                    G_prime.remove_edge(i,j)

    # remove self-loops
    G_prime.remove_edges_from(nx.selfloop_edges(G_prime))

    # remove disconnected vertices
    G_prime.remove_nodes_from(list(nx.isolates(G_prime)))

    if bipartite.is_bipartite(G_prime):
        W,N = nx.bipartite.sets(G_prime)
        W = list(W)
        N = list(N)
        return G_prime, [[int(G_prime.has_edge(W[i], N[j])) for j in range(len(N))] for i in range
(len(W))]
    else:
        return None, None
```

Encoding



```
'''
Encode data using a linear code whose parity-check matrix is the adjacency matrix
of a biregular (c,d) graph.
'''
def encode(data, c, d):

    B, A = generate_random_graph(len(data),c,d)
    if (B == A == None):
        return -1
    ~~A = utils.parmat_to_std_form(A)~~
    G = get_generator_matrix(A)

    # multiply the two matrices to get the codeword C
    return A, np.matmul(data,G)%2
```

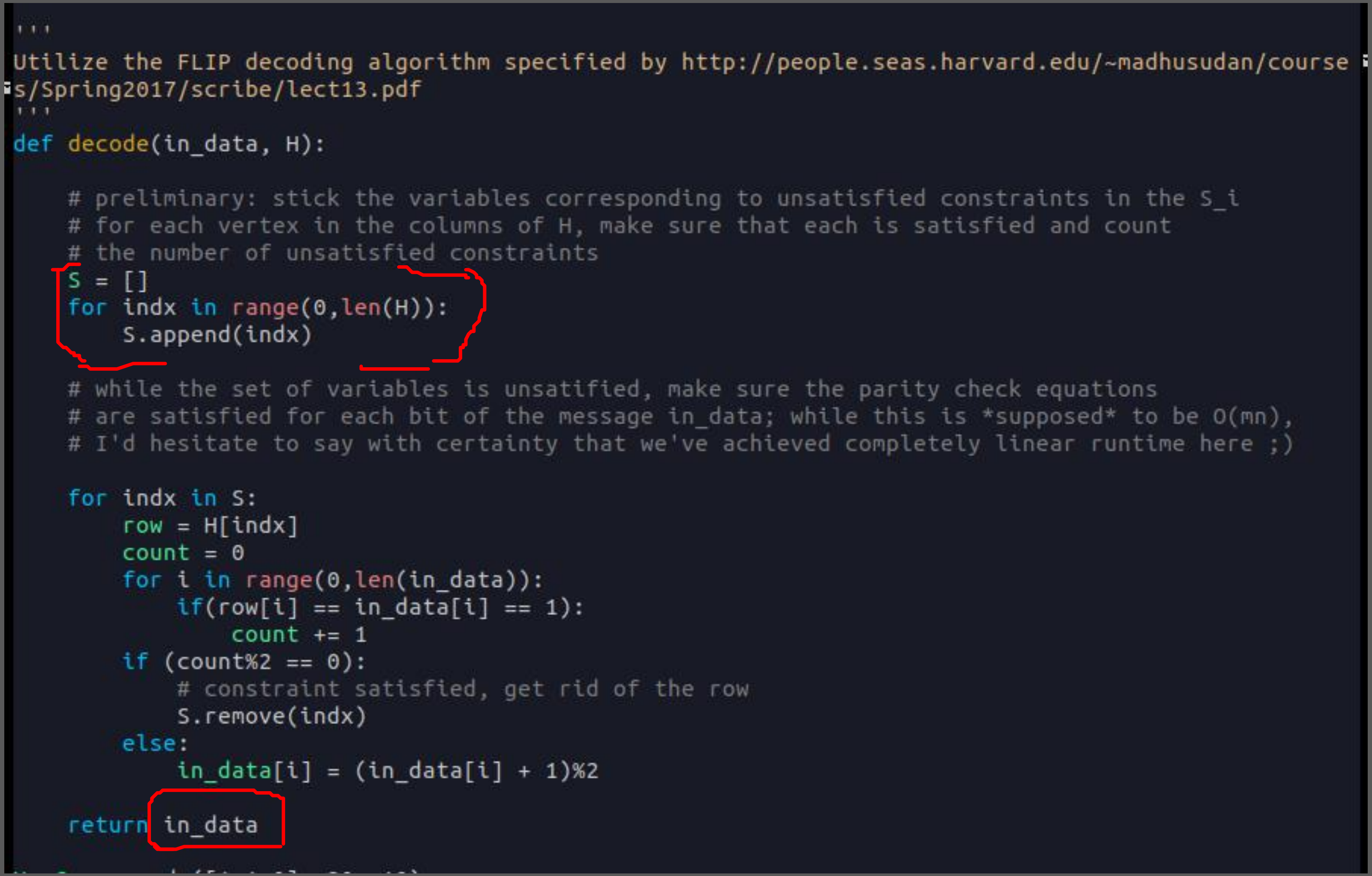




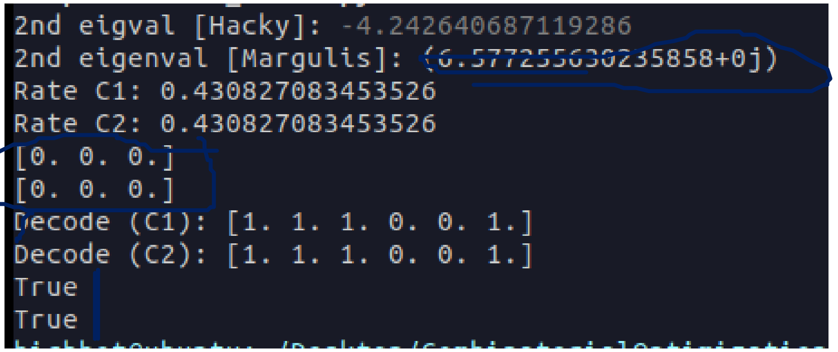
```
'''
Get the code by computing the generator matrix from the standard-form parity matrix H
'''
def get_generator_matrix(H):

    # get P^T, compute the transpose
    nrows = len(H)
    ncols = len(H[0])
    P_t = []
    for i in range(0, nrows):
        p_row = []
        for j in range(nrows, ncols):
            p_row.append(H[i][j])
        P_t.append(p_row)

    P_t = np.array(P_t)%2
    P = np.ndarray.transpose(P_t)

    # now concatenate the appropriate I.d. matrix
    n = P.shape[0]
```

$P^T | I_n$

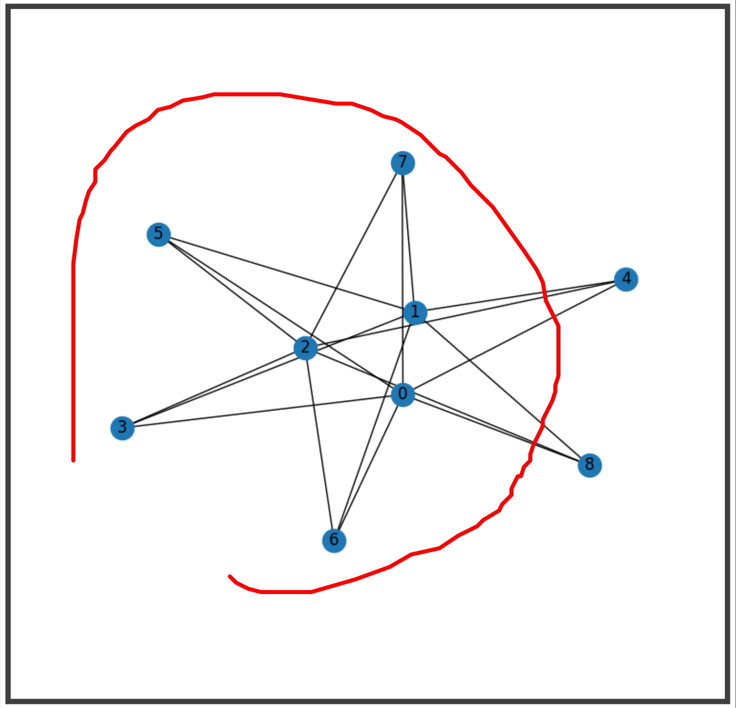
(Imperfect) Decoding

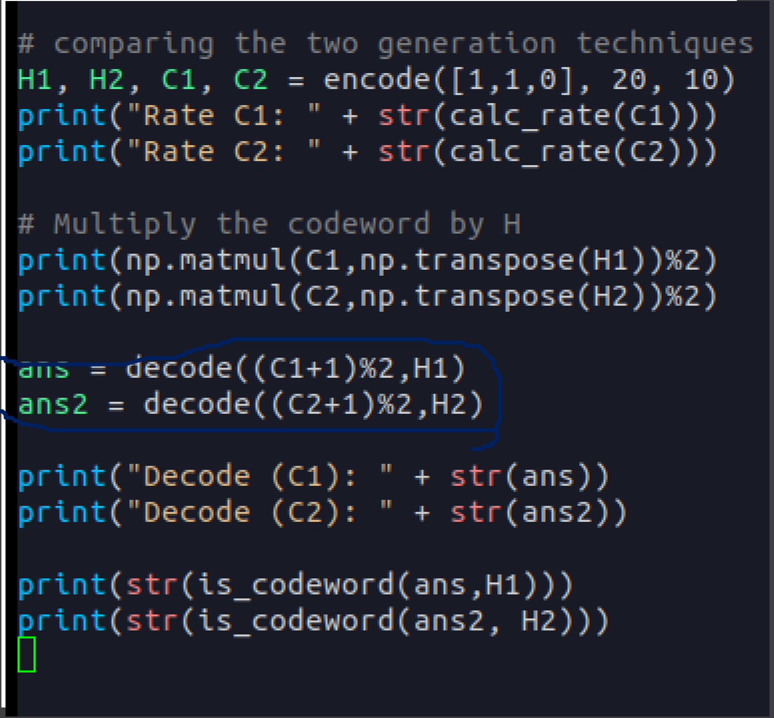


```
'''
Utilize the FLIP decoding algorithm specified by http://people.seas.harvard.edu/~madhusudan/course
s/Spring2017/scribe/lect13.pdf
'''
def decode(in_data, H):

    # preliminary: stick the variables corresponding to unsatisfied constraints in the S_i
    # for each vertex in the columns of H, make sure that each is satisfied and count
    # the number of unsatisfied constraints
    S = []
    for indx in range(0,len(H)):
        S.append(indx)

    # while the set of variables is unsatified, make sure the parity check equations
    # are satisfied for each bit of the message in_data; while this is *supposed* to be O(mn),
    # I'd hesitate to say with certainty that we've achieved completely linear runtime here ;)

    for indx in S:
        row = H[indx]
        count = 0
        for i in range(0,len(in_data)):
            if(row[i] == in_data[i] == 1):
                count += 1
        if (count%2 == 0):
            # constraint satisfied, get rid of the row
            S.remove(indx)
        else:
            in_data[i] = (in_data[i] + 1)%2

    return in_data
```

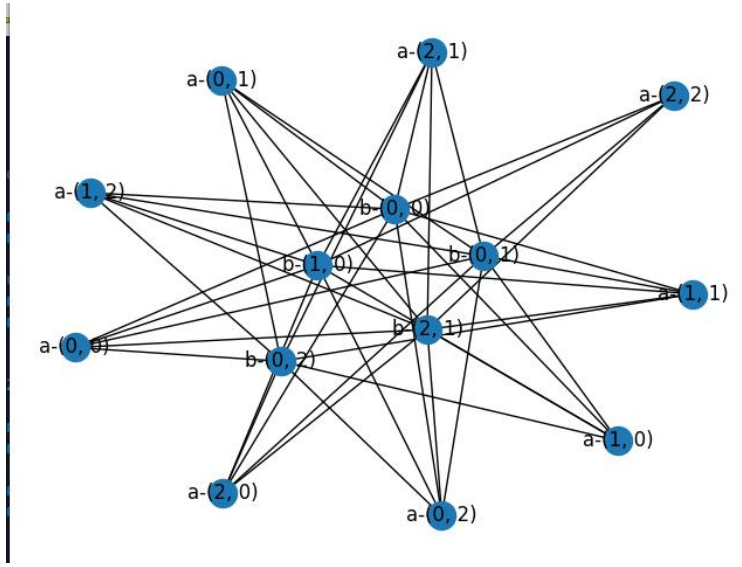


```
2nd eigval [Hacky]: -4.242640687119286
2nd eigenval [Margulis]: (6.577255630235858+0j)
Rate C1: 0.430827083453526
Rate C2: 0.430827083453526
[0. 0. 0.]
[0. 0. 0.]
Decode (C1): [1. 1. 1. 0. 0. 1.]
Decode (C2): [1. 1. 1. 0. 0. 1.]
True
True
```

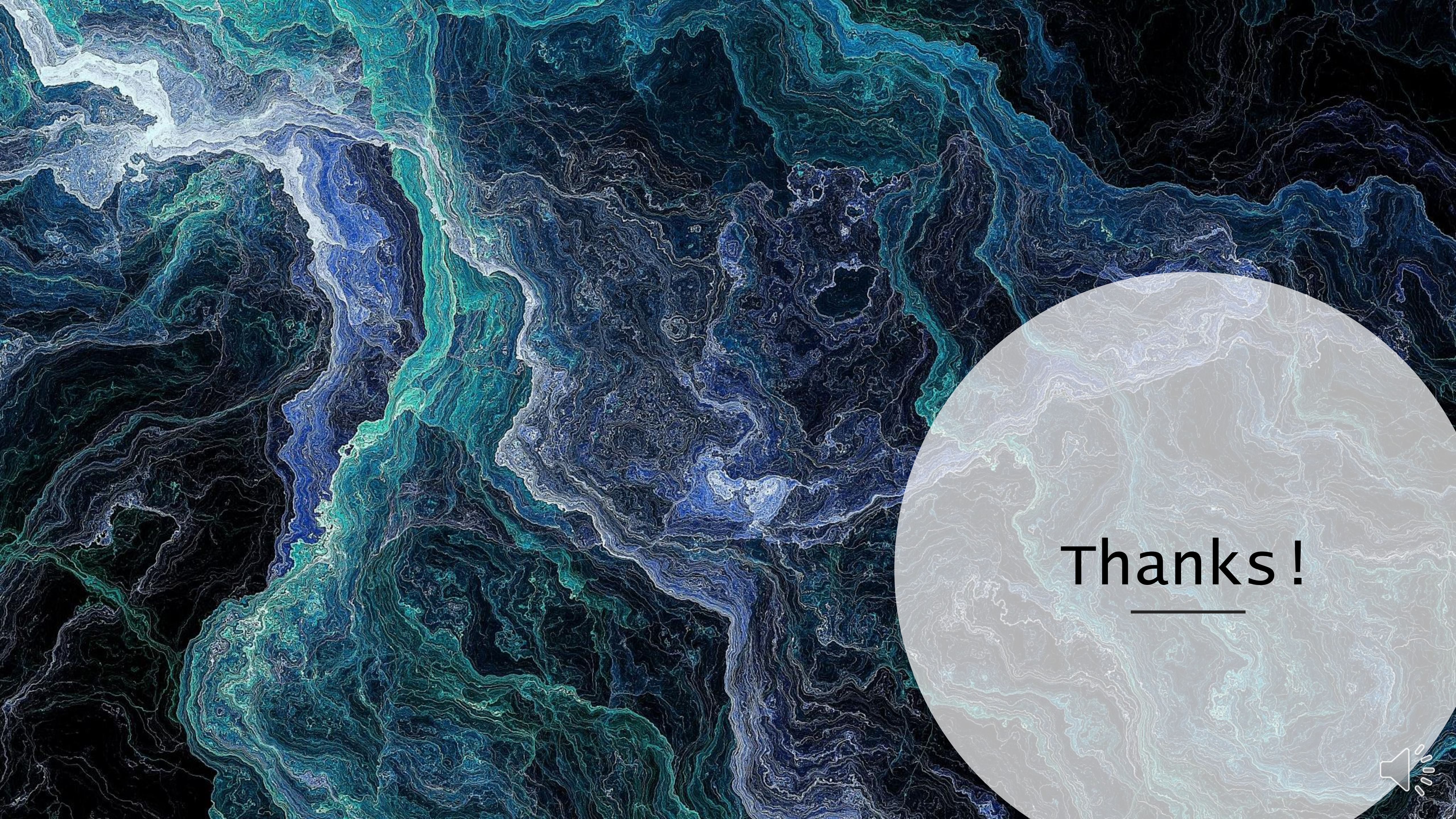




```
# comparing the two generation techniques
H1, H2, C1, C2 = encode([1,1,0], 20, 10)
print("Rate C1: " + str(calc_rate(C1)))
print("Rate C2: " + str(calc_rate(C2)))

# Multiply the codeword by H
print(np.matmul(C1,np.transpose(H1))%2)
print(np.matmul(C2,np.transpose(H2))%2)

ans = decode((C1+1)%2,H1)
ans2 = decode((C2+1)%2,H2)

print("Decode (C1): " + str(ans))
print("Decode (C2): " + str(ans2))

print(str(is_codeword(ans,H1)))
print(str(is_codeword(ans2, H2)))
```



Some Output



Thanks!

Bibliography

1. Krebs, Mike, and Anthony Shaheen. *Expander Families and Cayley Graphs*. New York, New York: Oxford University Press, 2011.
2. Barak, Boaz. ”Expander Graphs and their Applications.” Harvard University. https://www.boazbarak.org/expandercourse/allnotes.pdf
3. Spielman, Daniel A. ”Constructing Error-Correcting Codes from Expander Graphs.” Yale University. https://www.cs.yale.edu/homes/spielman/PAPERS/IMA.pdf.
4. Candes, Emmanuel, Mark Rudelson, Terrence Tao, and Roman Vershynin. ”Error Correction via Linear Programming.” University of Michigan. http://www-personal.umich.edu/ rudelson/papers/FOCS05.pdf.
5. Goldreich, Oded. ”Basic Facts About Expander Graphs.” https://www.wisdom.weizmann.ac.il/ ~oded/COL/expander.pdf.
6. Chow, Y.-T., Shi, W., Wu, T., and Yin, W., “Expander Graph and Communication-Efficient Decentralized Optimization”, arXiv e-prints, 2016.
7. Tao, Terrence. ”Basic Theory of Expander Graphs.” 254B: Notes 1, Basic Theory of Expander Graphs. Last modified December , 2011. https://terrytao.wordpress.com/2011/12/02/245b-notes-1-basic-theory-of-expander-graphs/
8. Spielman, Daniel A. ”Linear Time Encodable and Decodable Error Correcting Codes.” IEEEXplore. Last modified November 6, 1996. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=arnumber=556668.
9. Hoory, Shlomo, Nathan Linial, and Avi Widgerson. "Expander Graphs and their Applications." Bulletin (New Series) of the American Mathematical Society 43, no. 4 (October 2006): 439-561. https://www.cs.huji.ac.il/~nati/PAPERS/expander_survey.pdf.

Bibliography (Cont.)

10. Guruswami, Venkatesan. "Notes 8: Expander codes and their decoding." Carnegie Mellon University. Last modified March , 2010.https://www.cs.cmu.edu/~venkatg/teaching/codingtheory/notes/notes8.pdf.
11. Feldman, John, Tal Malkin, Roco A. Servedio, Cliff Stein, and Martin J. Wainwright. "LP Decoding Corrects a Constant Fraction of Errors." *IEEE Transactions on Information Theory* 53, no. 1 (January 2007): 82-89. https://people.eecs.berkeley.edu/~wainwrig/Papers/FeldmanEtAl07.pdf.
12. Spielman, David. "Properties of Expander Graphs." Yale University. Last modified October , 2015. https://www.cs.yale.edu/homes/spielman/561/lect15-15.pdf.
13. Sudan, Madhu. "CS 229r Essential Coding Theory. Lecture 13." Harvard SEAS. Last modified February , 2017. http://people.seas.harvard.edu/~madhusudan/courses/Spring2017/scribe/lect13.pdf.
14. Trevisan, Luca. "Stanford University — CS366: Graph Partitioning and Expanders." Stanford University. Last modified March 4, 2013. https://lucatrevisan.github.io/teaching/cs366-13/lecture13.pdf.

Margulis Construction Tutorial: https://abelprize.no/sites/default/files/2021-04/Margulis_construction_expander_english.pdf