

Construction of Topological Color Codes

A Project Report

submitted by

SAI MALI A

*in partial fulfilment of the requirements
for the award of the degree of*

MASTER OF TECHNOLOGY



**DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.**

JUNE 2014

THESIS CERTIFICATE

This is to certify that the thesis titled **Construction of Topological Color Codes**, submitted by **Sai Mali.A**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology and Master of Technology**, is a bona fide record of work done by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Place: Chennai

Date: 13-06-2014

Dr. Pradeep Sarvepalli
Assistant Professor
Dept. of Electrical Engineering
IIT-Madras, 600036

ACKNOWLEDGEMENTS

I take this opportunity to express my deepest and heartfelt gratitude to my project guide Dr. Pradeep Sarvepalli for his valuable guidance and motivation throughout the project. I am extremely grateful to him for his time to guide me during the project, I have learnt so much from him in the past year and I cannot thank him enough.

I express my gratitude to IIT Madras and the Electrical Engineering Department for these fruitful years. A special thanks to all the faculty and staff for the excellent environment and infrastructure they have provided over the years.

I would like to thank all my lab members with whom I had various insightful discussions. On a personal level, I am grateful to my friends(Mali, Swarun, Nikhil, Ragha, GS, Pop, Subi and V) who have grown with me all these years and being my constant moral support.

I am forever indebted to my parents for their unconditional love, support and guidance through all hardships. I dedicate this thesis to them.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	2
1 Introduction	5
1.1 Qubits	5
1.2 Code	5
1.3 Quantum error correcting codes	7
1.4 Color codes	8
1.5 Organization of Thesis	9
2 Background on Topological Color Codes	10
2.1 Quantum Information	10
2.2 Stabilizer formalism	11
2.2.1 Pauli Matrices and stabilizers	11
2.2.2 Effect of unitary gates on stabilizers	13
2.2.3 Performing measurements using stabilizers	14
2.2.4 Logical operators for Stabilizer codes	15
2.3 Fault Tolerant Quantum Computation	16
2.3.1 Fault Tolerance	17
2.3.2 Transversal gates	17
2.4 Color codes and Collexes	18
2.4.1 Color codes- the idea	18
2.4.2 Defining color codes-Stabilizers	19

3	Construction of Color codes	21
3.1	Constructing a 2-colex	21
3.1.1	The cell inflation algorithm	21
3.1.2	Implementation of the algorithm in <i>MATLAB</i>	21
3.2	Bipartite algorithm	30
3.2.1	Bipartite Algorithm	30
3.2.2	Implementation of the algorithm in <i>MATLAB</i>	31
4	Analytical aspect of Color code construction	36
4.1	Cell inflation algorithm for 2-colex	36
4.2	Cell inflation for a 3-colex	38
4.3	Bipartite algorithm for a 2-colex	48
4.4	Bipartite algorithm for a 3-colex	49
4.5	Scope for further work	51

CHAPTER 1

Introduction

1.1 Qubits

We are interested in using quantum systems to process information. The building block of quantum information is the quantum bit or *qubit* analogous to a bit in the classical sense. A qubit has two possible states, $|0\rangle$ and $|1\rangle$ where $|\rangle$ is a notation used in quantum computation. These states come from the **Hilbert space** which in our context is a finite dimensional vector space with a norm defined on it. Quantum bits differ from classical bits in the states it can attain.

Unlike classical bits, qubits can be in a linear combination of $|0\rangle$ and $|1\rangle$. This linear combination is called *superposition* of the two states $|0\rangle$ and $|1\rangle$. A general state $|\psi\rangle$ is given by $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ where $|\alpha|^2 + |\beta|^2 = 1$. This restriction arises because of the nature of α and β : they are probabilities. Suppose we wish to know the state of the qubit $|\psi\rangle$, it is enough to know α and β . But observing the state, a process called *measurement*, we will only ever get $|0\rangle$ with probability $|\alpha|^2$ or $|1\rangle$ with probability $|\beta|^2$. This is why we had the restriction on $|\alpha|^2$ and $|\beta|^2$: The probabilities must add up to 1, hence $|\alpha|^2 + |\beta|^2 = 1$.

1.2 Code

We would like to store information in quantum bits(qubits) and process them, sending them from one place to another while taking care of errors. The measure-

ment performed on a state is *irreversible*: Once we get $\alpha|0\rangle$ or $\alpha|1\rangle$ it is impossible to determine the original state. Hence, we would like to be careful about measurements happening in the qubits where we store information. But, quantum systems interact with the environment and we have a process called *decoherence*. Roughly speaking, this interaction with the environment causes a measurement to happen on the state and hence, the state will become one of either $\alpha|0\rangle$ or $\alpha|1\rangle$.

We wish to process information without losing any information. In information theory, we have a way of achieving this: *Code*. A code can be understood as a way of converting information from a source into some symbols. These symbols are then used for communication or storage. A code is nothing but an algorithm to perform this conversion. In order to use the information again, we need to get the information back from the symbols into a form we understand. The symbols themselves are called *codewords*, the process of converting given information into symbols is called *encoding* and getting back the information we have given into the symbols is called *decoding*.

Suppose we store information in the form of bits, 0 or 1. When we transmit this information from place to place through a medium (this is called *channel* in communication) or even while storing it somewhere, there could be an error occurring on the bits- A 0 could become a 1 or vice versa. A simple example of how a code works is by demonstrating the *repetition code*. Say we encode every 0 into 000 and every 1 into 111. Then, sending a 0 over a channel is simply sending 000 over the channel after encoding. Say a *bit flip error* occurs on the second bit, i.e. the second bit becomes a 1. Then, the received information is 010. However, 010 is not a valid codeword because the only form of codewords we could have are 000 or 111. Therefore, we know an error has occurred while transmitting information. This is called *error detection*. Now, without any knowledge of what was being

sent and assuming only one bit would have encountered an error, one can deduce that 0 was the bit sent. This is called *error correction* and this is more useful than error detection. This error correction is performed alongside decoding. The end result will only be the information we had originally, either 0 or 1 and it remains the same for the sender and receiver though some errors could have occurred.

This repetition code is an example of a *classical error correcting code*. It is **classical**, because classical bits 0 and 1 are used to store information and it is **error correcting** because it can ideally correct some errors occurring on the *encoded information* (ie, errors occurring on information after we have converted them into symbols/codewords). In the case of the repetition code we encountered, it can correct single bit errors. This is a simple way of storing and processing information stored in classical bits.

1.3 Quantum error correcting codes

Analogous to classical codes, we use *Quantum codes* to deal with quantum information and protect the information from noise. In quantum codes, we store the information in qubits instead of bits. But we have already seen in 1.2 about how qubits are different from bits and what happens if they interact with the environment.

We are interested in protecting quantum information from noise reliably and for this, we turn to the analogue of classical error correcting codes, namely *quantum error correcting codes* which use qubits to store information and perform error correction. The *noise* can occur during storage of quantum information, while transmitting/communicating information or when performing operations on the encoded information using gates.

One important aspect of creating codes is *computational complexity*. Most of the encoding and decoding operations are performed on computers and we often need to do this with finite time and resources. Crudely speaking, an operation having low complexity needs a lesser amount of time and resources (example, computing power) with a computer compared to one with high complexity. For example, if the complexity is a *polynomial in n* then it means that if n is some unit of the system (for example, the number of bits of a repetition code) then as we increase n to higher and higher values, the resources needed to perform the algorithm would increase as a polynomial in n .

While constructing a code to process information, one often sees that the decoding is the operation that has more complexity than anything else. Therefore we are looking for quantum codes which have low complexity of decoding.

1.4 Color codes

A subclass of quantum codes are called topological quantum codes. Topological codes are just quantum codes with a defined topological order on them. We are going to look at lattices that are embedded on surfaces of some genus and the qubits of the code will be placed on some cells of the lattice (vertices/edges). Then the checks will be performed for every face of the lattice.

Topological quantum codes are favored because of the way they encode information globally and hence, any local noise would not affect the code. Apart from this, the low complexity of decoding, as well as usage of very few qubits for performing checks makes these codes attractive to use. An analogy can be drawn to Low Density Parity Check codes (LDPC codes) which are widely used in classical coding for their low decoding complexity and ease of checks.

Color codes are also topological codes but there is a restriction on the type of lattice to be used for embedding in a closed manifold. A simple example of color code is a 3-valent, 3-face colorable lattice defined on a torus, where the qubits are placed on the vertices of the lattice. We will discuss this example in detail, emphasize why color codes are important, while describing an existing construction in [1] as well another approach proposed in [2]. We will implement both the constructions and extract the quantum code from them. Thereafter we will see how the construction in [1] is not general and will give only a restricted set of codes whereas the latter approach in [2] will give the most general color codes in lower dimensions.

1.5 Organization of Thesis

Chapter 2 deals with a background of topological color codes, with an overview of quantum information concepts, including Pauli operators, stabilizer formalism needed to introduce color codes.

Chapter 3 explains a procedure for the construction of color codes from a random lattice proposed by [1] and another construction proposed in [2]. We will implement these constructions for a random lattice and discuss the outcome.

Chapter 4 details the limitations of Bombin's construction for lower dimensional cases and explains how using another approach gives a general class of codes. We also present the scope for future work related to the problem.

CHAPTER 2

Background on Topological Color Codes

As we saw in 1, one big reason why topological quantum codes are favorable is because of the way information is encoded. By design, information here is stored *globally*, so any noise that is local does not affect the code. In this chapter, we will provide a background on topological codes, in particular color codes. We also introduce associated concepts that are useful to understand subsequent material.

2.1 Quantum Information

As we saw in the earlier section Quantum states are unlike classical states because a particular state in the former could be in a superposition of many classical states. Performing a measurement would only reduce it to one of these states and information is lost, because this process is not reversible. There is a phenomenon called *decoherence* that affects quantum states. Roughly speaking, this happens due to the system interacting with the environment which leads to a ‘collapse’ of a state to either $|0\rangle$ or $|1\rangle$ into the single qubit state. This collapse is akin to a measurement being performed on the state.

Protecting quantum information from noise reliably becomes our main concern and we turn to quantum error correcting codes (QECC) for this purpose. One should keep in mind that noise can arise both due to transmitting information as well as performing operations on the qubits by means of gates. Our job is to look

for codes that perform this reliably and with a low complexity. In order to understand Quantum error correcting codes, we will look at the stabilizer formalism below.

2.2 Stabilizer formalism

Just like classical codes, we are going to encode the given information by adding redundancy- so that sufficient information is present in the *encoded codeword* when it is subject to noise.

How does one describe a code? A code is composed of its *codewords* which are in fact, the encoded information so the most natural way of describing a code is through codewords. In Quantum codes, the codewords come from the finite dimensional Hilbert space(a generalization of vector space in n dimensions with a scalar called *inner product* that can be calculated for any two vectors in the space). However,the *Stabilizer formalism* provides us a nice way of describing the codes without using the codewords themselves, instead using a much smaller set. Let us begin understanding the stabilizer formalism by looking at Pauli matrices.

2.2.1 Pauli Matrices and stabilizers

The *Pauli matrices* I,X,Y,Z are defined for one qubit as follows:

$$\text{Pauli- I} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\text{Pauli- X} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$\text{Pauli- Y} = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

$$\text{Pauli- Z} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Why are Pauli matrices important? These matrices also act as gates performing various operations. For example, in the 1-qubit case say the Pauli X gate act on the state $\alpha|0\rangle + \beta|1\rangle$. A general state $\alpha|0\rangle + \beta|1\rangle$ is represented by a 2×1 matrix with first entry α and second entry β : $\begin{pmatrix} \alpha \\ \beta \end{pmatrix}$ Therefore, the state $|0\rangle$ would be

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

The Pauli matrices have certain nice properties that we exploit to great effect: They are Hermitian, unitary and involutory i.e two distinct Pauli gates always anticommute (Example $XY = -YX$). Interestingly, all the Pauli matrices have eigenvalues only ± 1 . This is useful later in describing stabilizer codes.

We are interested in the *Pauli group* on n qubits, which is the set of all possible n-qubit Pauli gates. Note that a general n-qubit Pauli matrix is nothing but the tensor product of n individual matrices, each of them coming from the set $\{ I, X, Y, Z \}$ along with scalars. For example, $G_1 = \{ \pm I, \pm iI, \pm X, \pm iX, \pm Y, \pm iY, \pm Z, \pm iZ \}$. A quick check will tell us that this forms a group under multiplication.

Now, we define what we call as *stabilizers* in algebra for our group G_n . Suppose we take a set of n qubit states S . Now, suppose we take a subgroup H of G_n such that every element of H fixes every element of S (The operation being tensor multiplication). Then such a set H is called the *stabilizer* of the vector space S . Stabilizers are useful because we instead of describing a quantum code by its codewords, we define the code by it's set of *stabilizer generators* which are nothing but the generators of the stabilizer set H .

As mentioned above, Pauli operators have eigenvalues only ± 1 . Since we are using a subgroup of the Pauli group in n dimensions to be the stabilizer generators, we now define the *codespace* (ie, the set of codewords) to simply be the $+1$ eigenspace of all the elements of the subgroup H (This is akin to *fixing* each codeword when operated by a member of H).

Therefore, we can take certain subgroups of the n qubit Pauli group and using this as the stabilizer generators to get a non trivial code space. It must be emphasized that these subgroups must obey two sufficient conditions to get a non trivial codespace: The elements must commute and $-I$ should not be an element of the stabilizer generators. The proof is quite simple and available in [3]

2.2.2 Effect of unitary gates on stabilizers

Now that quantum error correcting codes can be described using the stabilizer formalism, the natural step will be to figure out a way to understand how noise and operations on the code affect them. Explicitly, we would like to know the effect of such processes on the stabilizer generators themselves.

Let us say we have an operation that is *unitary*. A matrix U is said to be unitary if $UU^\dagger = I$ and one can easily check that the Pauli matrices are unitary themselves. Why are we interested in the effect of Unitary operators? We can consider noise to simply be the effect of a unitary operator on the codewords.

Say we have a stabilizer subgroup H that stabilizes a 2^n dimensional vector space S . Now, for any element h of H , a unitary gate U acting on an element of S $|\psi\rangle$ will yield $U|\psi\rangle = Uh|\psi\rangle = UhU^\dagger U|\psi\rangle$. This action is true for any element of S and H . We can conclude that for the vector space US , the stabilizer group will be nothing but UHU^\dagger . Thus we can describe the effect of error on the codewords

through the stabilizers themselves!

One important result in quantum error correcting codes is *linearity*. Suppose in the 1-qubit case we can correct say the errors due to the unitary gates $\{I, X, Y, Z\}$. Then any error will be a linear combination of the Pauli matrices and hence, can be shown to be corrected. [7]

2.2.3 Performing measurements using stabilizers

Say we perform a measurement $h \in G_n$ and without loss of generality we can assume h does not have a multiplicative factor of -1 or $\pm i$ (We can do this because h is a product of Pauli matrices). Assuming the system is in a state $|\psi\rangle$ with stabilizer $H = \{h_i, i = 1, 2, \dots, n\}$. We wish to see how performing the measurement h on the state affects the stabilizer, ie, we wish to know the stabilizers of the post-measurement state.

Suppose the measurement operator h commutes with all the elements of H . Then $h_j h |\psi\rangle = h h_j |\psi\rangle = h |\psi\rangle$ for each j which would mean $h |\psi\rangle$ is in S and would be a multiple of $|\psi\rangle$. Because the Pauli matrices are involuntary, $h^2 = I$ tells us $h |\psi\rangle = \pm |\psi\rangle$. Therefore, either h or $-h$ is in the stabilizer.

Say h belongs to the stabilizer. Hence $h |\psi\rangle = |\psi\rangle$ which leads us to conclude that such a measurement leaves both the state of the system and the stabilizer unaltered. Similarly say h anti-commutes with one of the elements of the stabilizer say g_1 then it can be shown [3] that post measurement, stabilizer groups $\{\pm h, h_2, \dots, h_n\}$

2.2.4 Logical operators for Stabilizer codes

Suppose we were dealing with n qubits and a given stabilizer group H that has been generated by r elements, one can see this specifies a subspace of dimension 2^{n-r} which is used to encode $k = n - r$ qubits. [6]

Now, the *normalizer* N of a H in G_n is defined as the set of operators $G \in G_n$ such that $GhG^\dagger \in H$ for every $h \in H$. Since all the elements of the stabilizer are unitary and commute with each other, we can say that the stabilizer H is automatically in the Normalizer N of the stabilizer set. We are interested in the set of operators in the Normalizer but not in the Stabilizer ie, the set $N - H$ because these will represent the k Pauli operators on the code.

Why are normalizers important? Remember that in 2.2.1 we mentioned that any two distinct Pauli gates either commute or anticommute and taking this in n dimensions we get the same result. Suppose we have an error E that is also from the Pauli group G_n and say E anticommutes with the element h of the stabilizer (which also comes from the Pauli group in n dimensions). Now, after the error the codeword ψ becomes $E\psi$ but if we apply the operator h on this, we get $hE\psi = -Eh\psi = -E\psi$ (Because h is in the stabilizer of ψ). One can clearly see that we get an eigenvalue -1 when we apply the eigenvector h on the error. But the codewords are in the $+1$ eigenspace of the stabilizer. Thus, measuring the eigenvalue is a simple way to detect errors which anticommute with elements of the stabilizer.

One key takeaway about the stabilizer is that apart from being a reference point for *defining* the set of codewords, they are also the means of performing *checks* on the codeword.

2.3 Fault Tolerant Quantum Computation

Quantum computation entails us to perform operations on states that have been encoded using a particular coding mechanism. There could be errors when gates are used for such operations and we do not want a few errors(say on some qubits) to become widespread and affect a large number of the remaining qubits. The simplest example to illustrate this is the CNOT gate, which is a common gate used in quantum computers. CNOT is a 2-qubit gate. In simple terms, CNOT acts on two qubits as input. If the first qubit is 1, it flips the second. If the first qubit were zero, the second is unchanged. The matrix form of the gate can be given as:

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

Now suppose we use the CNOT operation say on encoded qubits. Suppose the first qubit had an error in the input, then the second qubit is flipped. Hence, we have *two* qubits at the output of the gate that suffer from error even though only one of them was in error initially. If the same scenario happens in many gates, we would have a big chunk of the encoded qubits that are error-prone due to a smaller number of qubits in error.

To avoid situations like the above, we introduce what is known as fault-tolerant quantum computation, as discussed below.

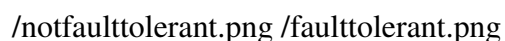
2.3.1 Fault Tolerance

When trying to build a circuit to implement a quantum error correcting code, we would like to protect the information from errors, especially those errors while passing the information through the various gates in the circuit. In the beginning of the circuit before encountering gates, we could encode the information and decode it before every gate so that the gate operation is performed and re encode it after every gate. But this would not protect the information from errors due to the gate itself.

We want gates to act directly on encoded states but correctable errors from one gate should not propagate without limitations to the rest of the circuit. This means we would implement the gates on encoded states in such a way that we don't lose information if one gate produces an error. For a given error-correcting code, a fault-tolerant implementation of a gate would restrict the evolution of the error so that correctable errors do not propagate leading to uncorrectable errors.

2.3.2 Transversal gates

To do this, we use Transversal gates: The i -th qubit in a block of a QECC interacts only with the i -th qubit of other blocks and therefore, error in this qubit will not affect more than one qubit in another block, hence fault tolerant.

 /notfaulttolerant.png /faulttolerant.png

Therefore, the idea of fault tolerant computing is to get an appropriate quantum code such that the gates required can be implemented *transversally* (or more generally, in a fault-tolerant manner). We also require the measurements performed to be done in the same manner. This is where the idea of color codes are attractive

to us- there are a variety of gates than be implemented as a transversal gate, as we shall see in the next section.

2.4 Color codes and Collexes

2.4.1 Color codes- the idea

The basic idea of a topological code is as follows: Physical qubits of the code are placed in a lattice and they interact only with nearby qubits. We now embed this lattice in a closed manifold of some genus.

Complexes are a way of giving combinatorial structure to a D-dimensional manifold. The manifold is divided into a hierarchy of objects of increasing dimension: points(0-cells), lines(1-cells), faces(2-cells) etc. D- collexes are complexes in D-manifold which is D+1-valent and colored with D+1 colors and we obtain color codes by building these collexes. In particular, we are interested in lower dimensions. In the 2-dimensional case, we have 0-cells to be *vertices*, 1 – *cells* to be edges and 2 – *cells* to be faces of a graph.

Color codes use a specific kind of lattice: 3-valent and 3-face colorable. *3-valent* means that for all the vertices of the graph, there are exactly 3 edges incident on the vertex, ie, the *degree* of each vertex is 3. *3-face colorable* means that if we color all the faces of the graph from a set of three colors, with the property that any two *adjacent* faces (faces whose boundaries share an edge in common) are always have different colors. This lattice is embedded in a surface of genus 1, which is like a torus. Let us define the toric color code and to do that, we use the stabilizer formalism.

2.4.2 Defining color codes-Stabilizers

Recall the stabilizer formalism discussed in 2.2. The same idea is being applied to color codes so that instead of having to describe codewords explicitly, we point out the stabilizer generators (we only need the independent generators) and the codeword can be put forward simply as the $+1$ eigenspace of all the elements of the stabilizer.

The qubits in the color code are placed at the vertices of the lattice. Now to get the elements of the stabilizer generators, we define two such elements per face of the lattice, as follows:

$$X_f = \prod_{v \in \text{face}} X_v$$

$$Z_f = \prod_{v \in \text{face}} Z_v$$

where X_v denotes the X Pauli operator acting on the qubit placed at the vertex v . The product here is a tensor product, so essentially these two operations perform a X or Z on every qubit of every face and this is done for all the faces of the lattice embedded on the torus.

Notice that we have $2F$ number of such operators in the graph where F is the number of faces of the graph. But all of them are not independent. For instance, say we color all the faces with three colors and call the set of faces colored i as F_i , say $i \in \{r, b, g\}$. Now, looking covering all the faces of the graph by color and using the definition of tensor products, we get

$$\prod_{v \in F_r} X_v = \prod_{v \in F_b} X_v = \prod_{v \in F_g} X_v$$

and also the same for Z

$$\prod_{v \in F_r} Z_v = \prod_{v \in F_b} Z_v = \prod_{v \in F_g} Z_v$$

We have two operators dependent on the rest in each of the two equations and hence, four among the $2F$ operators are dependent. Therefore, the number of independent stabilizer generators will be $g = 2(|F_r| + |F_b| + |F_g| - 4)$.

CHAPTER 3

Construction of Color codes

3.1 Constructing a 2-colex

3.1.1 The cell inflation algorithm

An algorithm has been proposed in [1] for constructing a D-colex in closed manifolds from an arbitrary complex. We discuss the same for constructing a 2-colex below. Given a graph embedded in a torus and three colors (say $\{r, b, g\}$)

1. Color each face of the embedding using one of the colors using one of the colors $x \in \{r, b, g\}$
2. Split each edge of the graph into two edges, both of which have the same endpoints as the original edge. Now color the new faces that are created between these two new edges with a different color in $\{r, b, g\} - x$
3. Transform each vertex, which is of say, degree d into a face containing d edges and color it with the remaining color in $\{r, b, g\}$

3.1.2 Implementation of the algorithm in *MATLAB*

The cell inflation algorithm requires us to start from a lattice embedded in a torus(a surface of genus 1) and proceed to build a 2-colex using the above steps. We implemented the algorithm in *MATLAB* for a graph embeddable on a torus and we

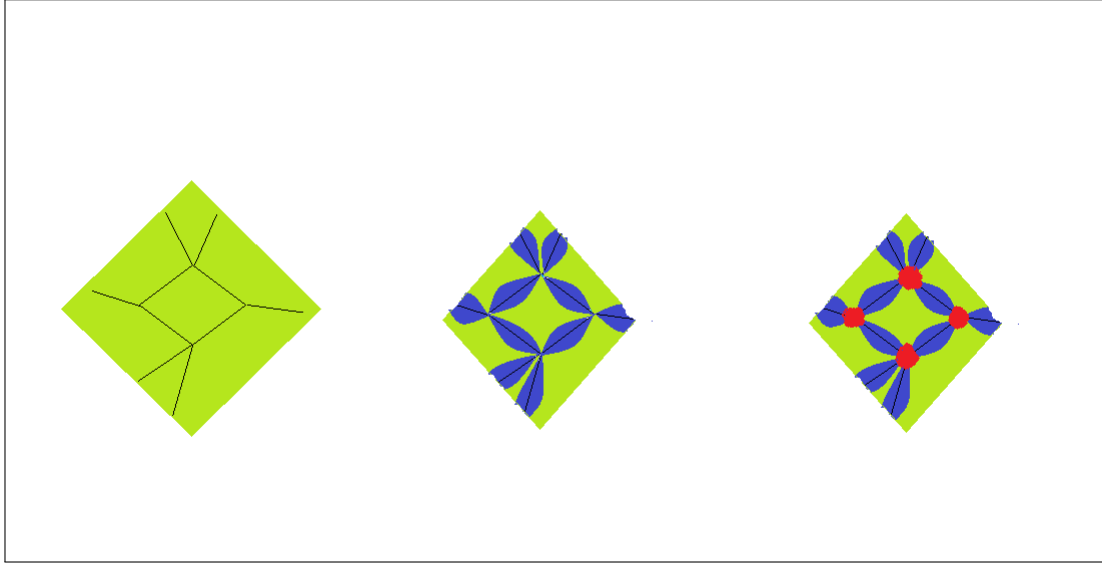


Figure 3.1: (a) All the 2-cells are colored green(b) Colored blue are faces derived by splitting the edges (c) 0-cells(vertices) are inflated to faces and colored red

generated the stabilizer matrix of the resultant color code arising out of the 2-colex.

Our implementation consisted of two parts:

1. **Voronoi tessellation:** In order to get a graph in 2-D that was embeddable(which means we need graphs where the edges do not intersect and it should be on a torus) we took the input to be the number of points *poin* for starting the Voronoi tessellation. *poin* denotes the number of points inside the unit square $[0, 1] \times [0, 1]$ and voronoi tessellation was performed for these points such that the resultant graph was an embedding on the torus.

Input : *poin* , the number of points needed to start Voronoi tessellation inside $[0, 1] \times [0, 1]$

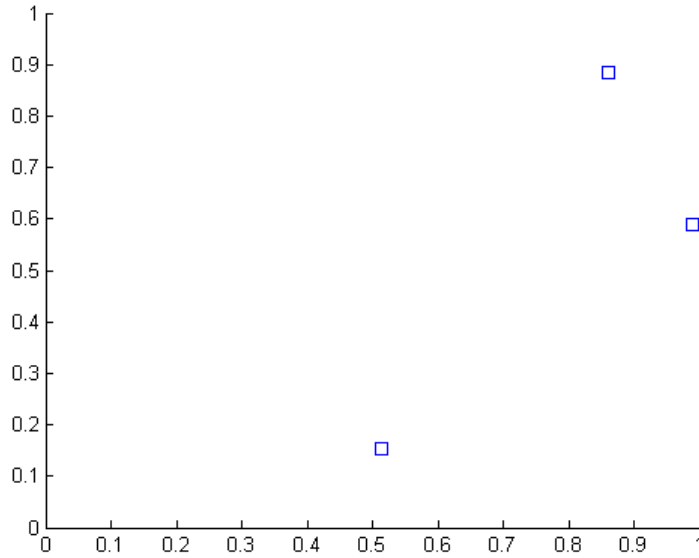
Output : A graph embedded in a torus. The graph is completely charac-

terised by

- (a) The **vertex coordinates** *vert* for the vertices of the tessellation inside $[0, 1] \times [0, 1]$
- (b) The **adjacency matrix** *full_voronoi_adj* which gives the adjacency matrix with the vertices being that of the resultant Voronoi diagram. Note that we have generated it as an embedding on a torus.
- (c) The **face data** *face_data* for the tessellation in which each row is a face and its elements are the set of vertices which make up the face in a clockwise or anticlockwise order.

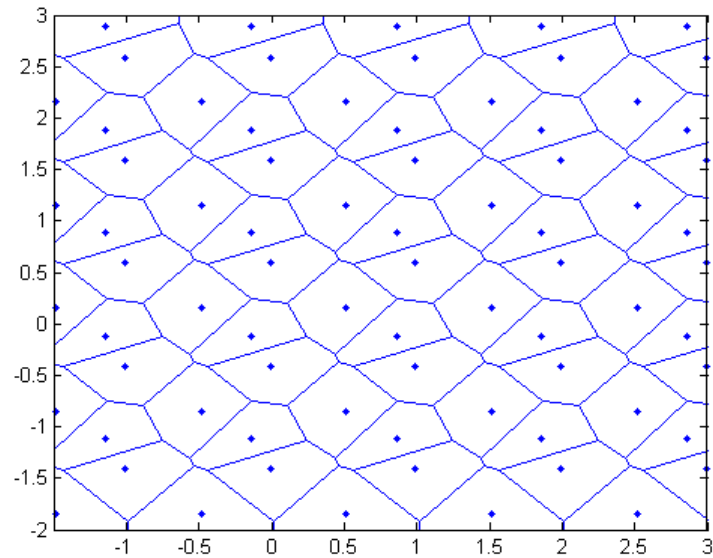
Method: We outline a rough pseudocode below. The full code is available in the Appendix of this thesis.

- (a) Take input *poin* and create random coordinates for these vertices.

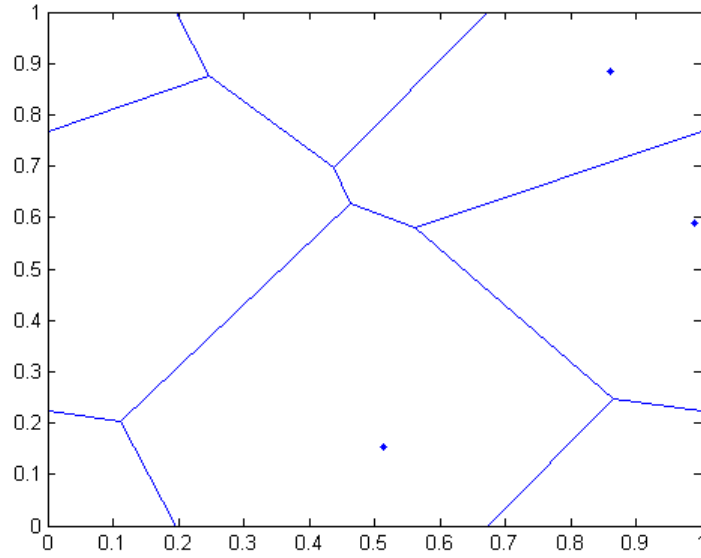


- (b) Now, using these coordinates generate Voronoi diagram inside the unit square using the inbuilt function *voronoi* for *poin*=3. The problem

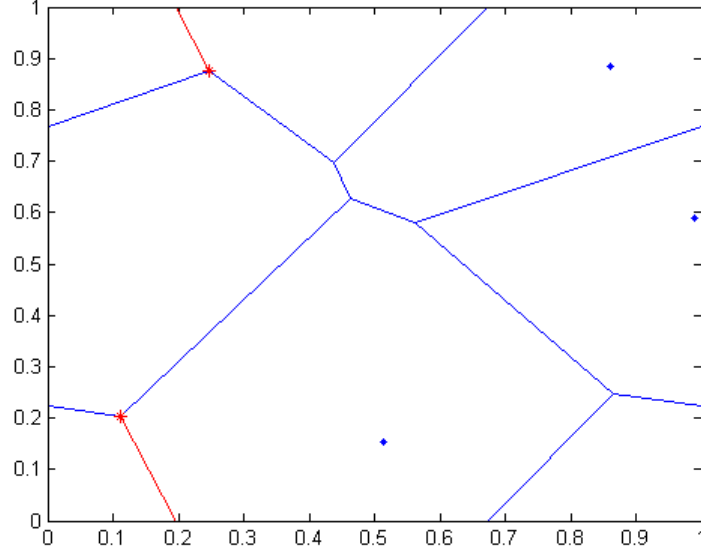
here is that we need an embedding inside a torus but the voronoi diagram will only be created for a set of points in a plane. Therefore, we make copies of the points in the unit square and tile them around the unit square to get a bigger $[-2, 3] \times [-2, 3]$. We now generate the voronoi diagram for these bigger set of points.



However, as an embedding we only need the points inside $[0, 1] \times [0, 1]$. Notice that this is an embedding on a torus.



- (c) Extract face information(order of vertices in every face) and vertex information(coordinates of the resultant vertices in the Voronoi diagram using the inbuilt function *voronoin*
- (d) To get the adjacency matrix, starting from every face i , take the order of the vertices specified. For example, if vertex j is followed by the vertex k in the face i then we know that j and k are adjacent vertices and hence, the entries corresponding to (i, j) and (j, i) will be 1 in the adjacency matrix.
- (e) Notice that this is more complicated than it seems because we perform the above procedure for the voronoi diagram inside $[-2, 3] \times [-2, 3]$ and not $[0, 1] \times [0, 1]$. Hence,there should be a procedure to get the vertices as an embedding and our code does it perfectly. Final output:



This looks similar to the previous figure but here, the vertices are an embedded in a torus and hence vertex i is considered to be adjacent to vertex j via the highlighted edge.

We now use this embedding and all the associated information to implement the Cell inflation algorithm for a lattice embedded on a torus.

2. **Generating the 2-colex:** Using the above embedding and given information, we implement Cell inflation algorithm step-by-step as discussed in 3.1.1

Input : The graph generated above and the information extracted: Adjacency matrix *full_voronoi_adj*, coordinates of vertices *vert* as well as the information about faces *face_data* though we do not need the third and can always be computed from the above two.

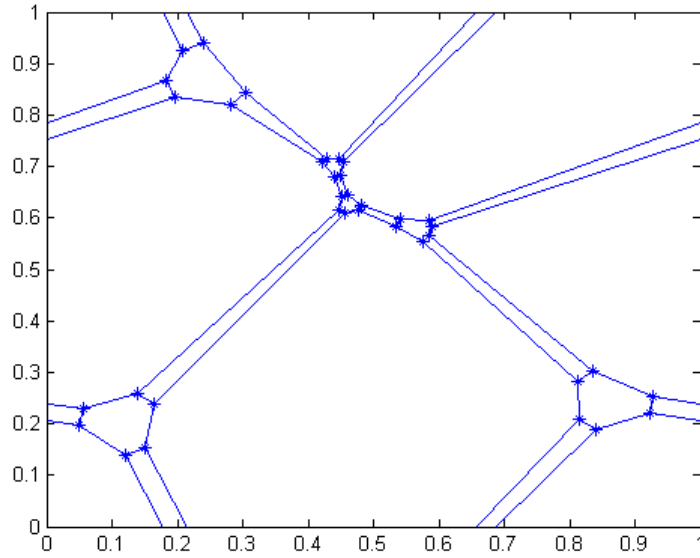
Output :

- (a) A 2-colex that is embedded on the torus. We also have the coordinates

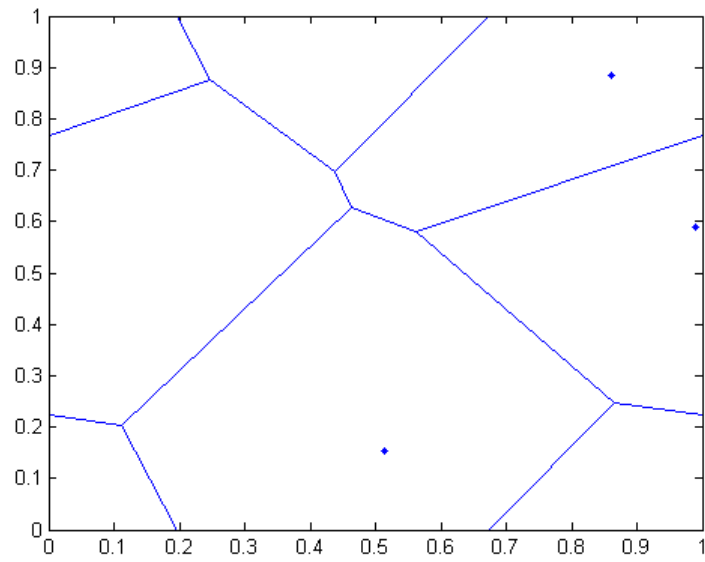
of the vertices, adjacency matrix and the order of the faces for the resultant 2-colex.

- (b) The stabilizer matrix of the resultant 2-colex. This will be a $f \times v$ matrix where f and v are the number of faces and vertices of the generated 2-colex respectively. If the vertex v occurs in the face f then we have the element on f th row and v th column to be 1 otherwise 0.

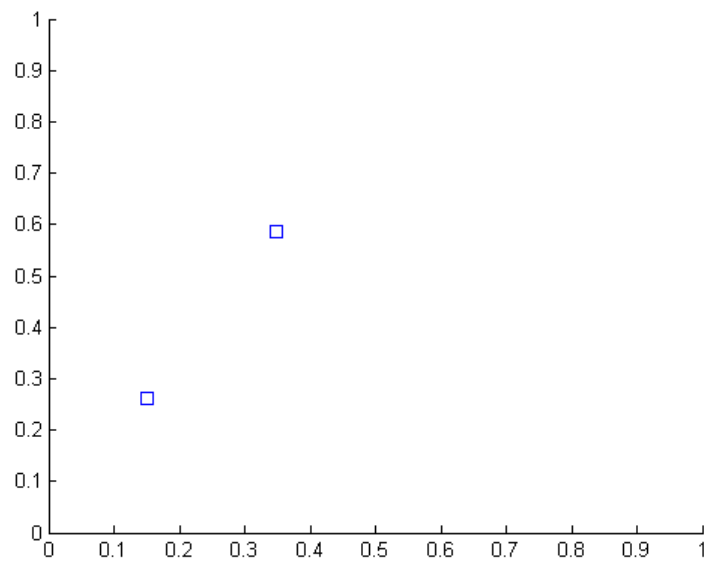
Method: The method follows the steps described in 3.1.1 but implemented slightly differently. We inflate the edges and vertices simultaneously, ie given an edge of the graph, each vertex among the two endpoints of that edge is split into two vertices and joined appropriately (This is what happens when a edge is inflated and vertex split). Then at each vertex v , we connect the new vertices created for each edge incident on v appropriately such that we get a polygon. The 2-colex for the given lattice is shown below:

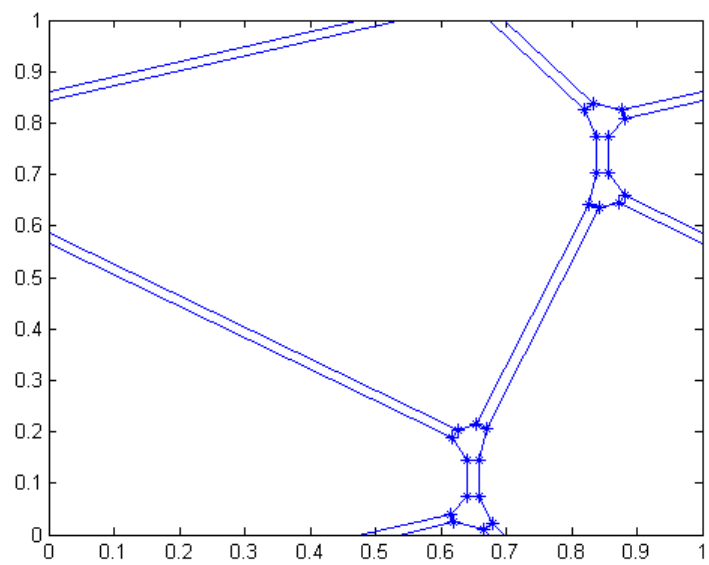
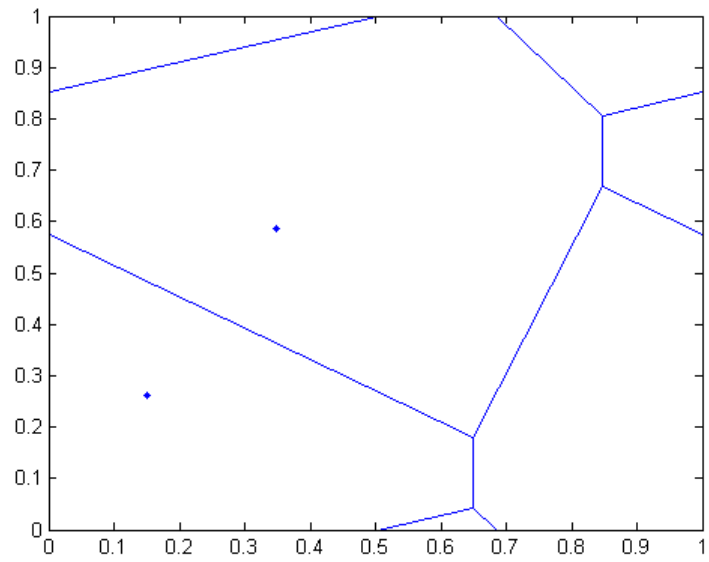


Compare it with the lattice:



We illustrate the 2-colex for another lattice, say for $poim = 2$:



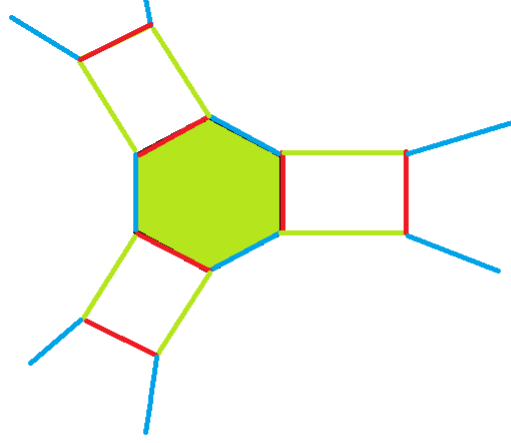


3.2 Bipartite algorithm

3.2.1 Bipartite Algorithm

Another algorithm for generating a 2-colex was proposed in [2] where the input is an arbitrary graph G that is bipartite and embedded on some surface. The algorithm outputs a 3-valent, 3-face colorable graph (that we know as the 2-colex), as illustrated below.

1. Take the dual of the bipartite graph G embedded on some surface. Say the dual is G^d , now G^d will be 2-face colorable (because G was bipartite).
2. Inflate every vertex of G^d . This is done such that, suppose a vertex has degree d , this is inflated to a face with d edges on its boundary. This are the faces colored green in the figure.
3. Notice that each of these new vertices have degree 3 and hence the graph is 3-valent. Before vertex inflation, the graph was 2-face colorable and coloring the face that we get by inflating the vertex with a third color, the new graph becomes 3-face colorable. Hence, this is a 2-colex.



3.2.2 Implementation of the algorithm in *MATLAB*

We are going to implement the Bipartite algorithm for an input that is an finite,arbitrary,bipartite graph embedded in the plane(surface of genus 0). We will look at graphs that are bipartite but having a finite number of vertices and consider this graph to be in the plane.The algorithm was implemented in *MATLAB* and we generated a 2-colex as well as the stabilizer matrix, as discussed below:

Notation : The arbitrary bipartite graph is \mathbf{G} embedded in the plane, dual of \mathbf{G} is \mathbf{D} . Say v is the number of vertices and f be the number of edges of \mathbf{G}

Input : The input we need is the parameters of \mathbf{G} :

1. *adj_mat* the adjacency matrix of the vertices of the graph(This will provide the information about the edges of the graph). This will be of size vv .
2. *face_info* Face information of \mathbf{G} . This can either be provided as the input or can be extracted from *vert_set* and *adj_mat*, but we require it in a particular format: The j th row of the matrix *face_info* will be the set of vertices com-

posing the face j in either clockwise or anticlockwise order. This matrix will have f rows. The number of columns will be the maximum number of vertices on the boundary of a face, taken among all f faces of the graph. For those rows(faces) with lesser number of vertices on the boundary, the remaining entries are zero. For example if the face i has 4 vertices(Say 10,11,13,14 in that sequence) on its boundary and the maximum number of vertices in any face is 6, then row i will have its entries to be 10,11,13,14,0,0 in the same order. While considering face i anywhere in the program, we will only consider the non zero vertices.

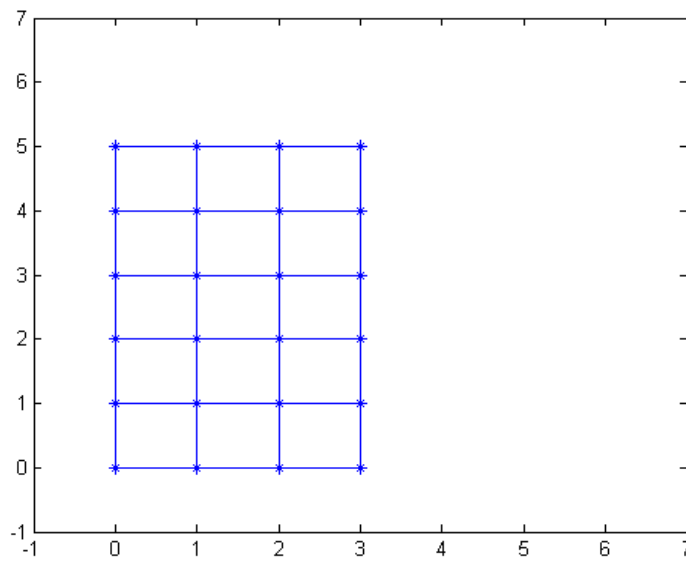
Output :

1. A 2-colex that is embedded on the plane. The 2-colex will be finite and the rest of the plane will be considered as the infinite face. We also have the coordinates of the vertices, adjacency matrix and the order of the faces for the resultant 2-colex.
2. The stabilizer matrix of the resultant 2-colex. This will be a $f \times v$ matrix where f and v are the number of faces and vertices of the generated 2-colex respectively. If the vertex v occurs in the face f then we have the element on f th row and v th column to be 1 otherwise 0. This will be the stabilizer matrix for the generated 2-colex.

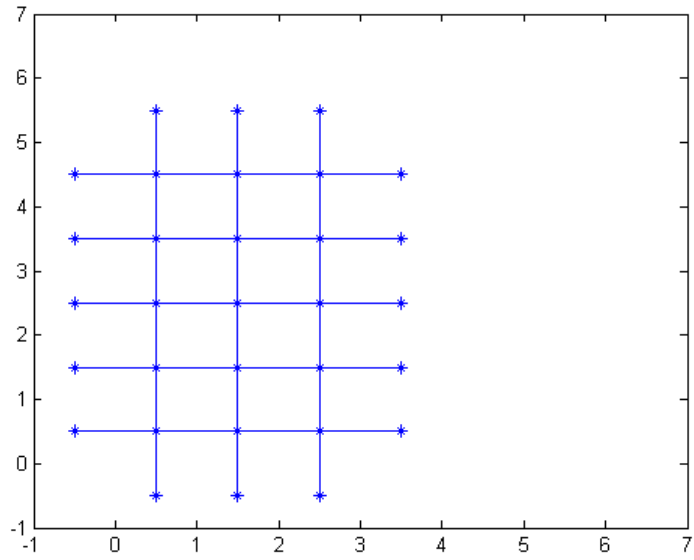
Method: Again, the algorithm is implemented as discussed in [2]. We take the input information of the given bipartite graph G embedded in the plane and use it to construct its dual. In the code, we get the vertex coordinates, adjacency matrix and face ordering of the dual graph D . After constructing the dual, we blow up the vertices of the dual to form the 2-colex. Note that we get all the required information(vertices,faces and adjacency matrix) of the resultant 2-colex

and use this to generate the stabilizer matrix of the 2-colex. Here is an illustration for a **square grid** of dimension 3×5 which is an example of a bipartite graph embedded in the plane:

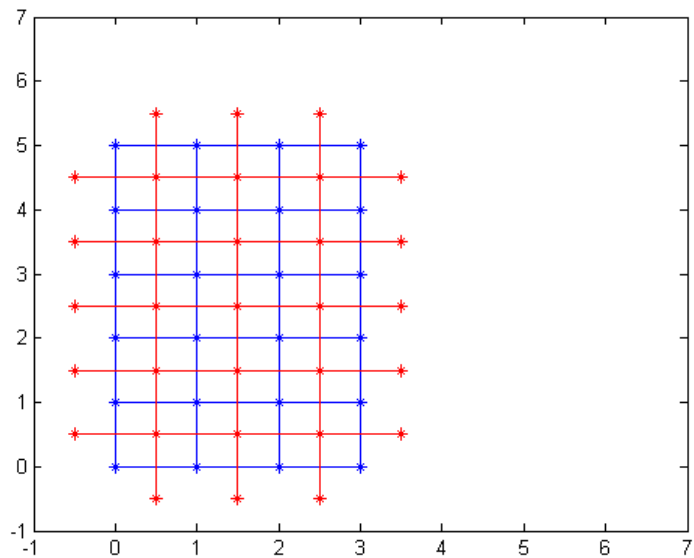
Input graph G for a simple square grid in the plane is shown below. The vertices are marked with $*$ and the input information is extracted for running the algorithm.



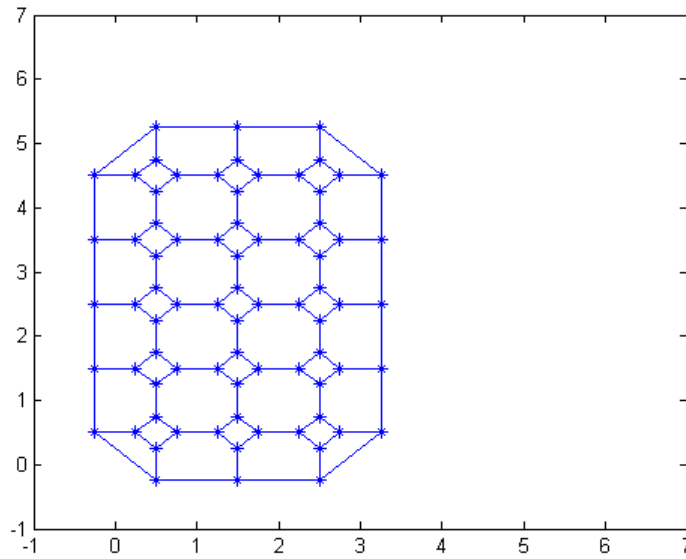
The rest of this plane not inside any face is considered to be the infinite face of G . The first part of the algorithm generates the dual D of the given graph G . In the case of the square grid shown above the dual D is given by:



Notice that if G had (v, e, f) as the parameters of the graph (number of vertices, edges and faces respectively) then the dual graph D has (f, e, v) as the parameters. Now we show both the graph G (colored blue) and its dual D (colored red) in the same figure:



Notice that all the red vertices(vertices of the dual that are not inside any face) represent the dual vertex corresponding to the infinite face of the original graph G . All these points actually denote only one vertex, namely the dual vertex of the infinite face. Now from the dual graph G we blow up the vertices and get the 2-colex. For the square grid, the 2-colex looks like:



This will be a 3-valent, 3-face colorable lattice as expected. We extract the coordinates of the vertex, adjacency matrix and face ordering for the 2-colex. Say the generated 2-colex has v' vertices and f' faces. Then the stabilizer matrix is nothing but a $f' \times v'$ matrix such that the element in the i th row and j th column $a_{ij} = 1$ iff face i has the vertex j in its ordering, else $a_{ij} = 0$. This gives us the information about the topology of the lattice and we can define the stabilizers of this code from the stabilizer matrix.

CHAPTER 4

Analytical aspect of Color code construction

4.1 Cell inflation algorithm for 2-colex

We discuss some properties of the cell inflation algorithm of [1] and implemented in 3.1.2. Taking the case of constructing a 2-colex from a random graph embedded in a torus, the algorithm does not yield all possible 2-colexes. To picture this, consider the 2-cells that we get by inflating a 1-cell. After getting the 2-colex, these faces (all of which are colored with the same color) are always 4-sided as noted in [2]. A short proof is presented below.

Lemma 4.1.1 *The 2-cells of the resultant 2-colex which are derived by inflating 1-cells as per the Cell inflation algorithm (discussed in 3.1.1) are always 4-sided.*

Proof : Take an arbitrary edge of the graph e . Let the boundary of e be the vertices (u, v) . Now in the edge inflation step of the algorithm, the edge e is split into two edges e_1 and e_2 such that both e_1 and e_2 share the same boundary (u, v) . Call the face created by u_1 and u_2 as f . Now f is a face with two edges e_1 and e_2 and two vertices on the boundary a and b . In the next step of the Cell inflation algorithm, the vertices of the graph (including a and b are blown up into 2-cells ie, faces).

This inflation of the vertices occurs such that the following property is obeyed: For a given vertex of the graph ν , say ω is a vertex adjacent to ν . Then the edge f

shared by ν and ω would be split into two, say f_1 and f_2 in the edge inflation step of the algorithm. Then in the vertex inflation step, two new vertices ν_1 and ν_2 are created instead of ν and the edges f_1 and f_2 would now be incident on ν_1 and ν_2 respectively instead of ν . And a new edge α is created between ν_1 and ν_2 . This happens for every vertex ν of the graph.

Now, we had e_1 and e_2 between two vertices a and b before vertex inflation. The face f had exactly two edges f_1 and f_2 with two vertices in the boundary a and b . After vertex inflation step has been performed for all the vertices of the graph (in particular a and b), a will split into two vertices a_1 and a_2 such that e_1 is incident on a_1 and e_2 is incident on a_2 . A new edge e_a is created between a_1 and a_2 . Similarly, b will split into two vertices b_1 and b_2 such that e_1 is incident on b_1 and e_2 is incident on b_2 . A new edge e_b is created between b_1 and b_2 .

Now, let us look at the face f after the algorithm has been performed completely. It has four vertices on its boundary, namely $\{a_1, a_2, b_2, b_1\}$ and four edges $\{e_1, e_2, e_a, e_b\}$. Thus, we can clearly see that this face f in the 2-colex is always 4-sided irrespective of the edge e that we start with. ■

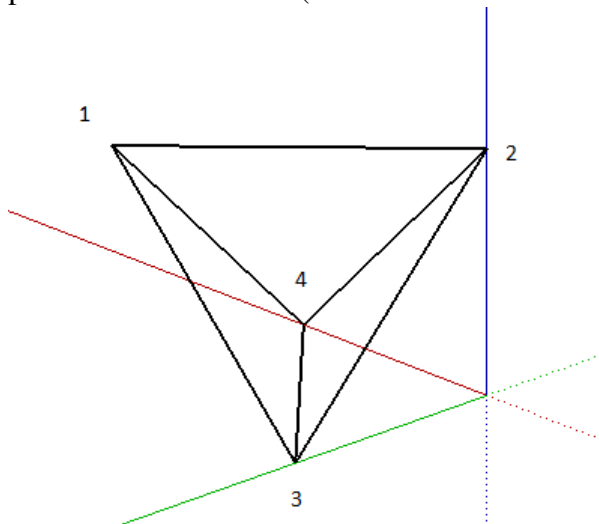
Therefore we see that the Cell inflation algorithm for generating a 2-colex may not yield any arbitrary 2-colex. This leads to the natural question of whether the same applies to 3-colexes as well. Given an arbitrary graph in 3D space and implementing the Cell inflation algorithm to get a 3-colex, whether the set of generated 3-colexes is a subset or equal to the set of all possible 3-colexes. We suspect that this is indeed the case and we visualized this restriction in 3-D

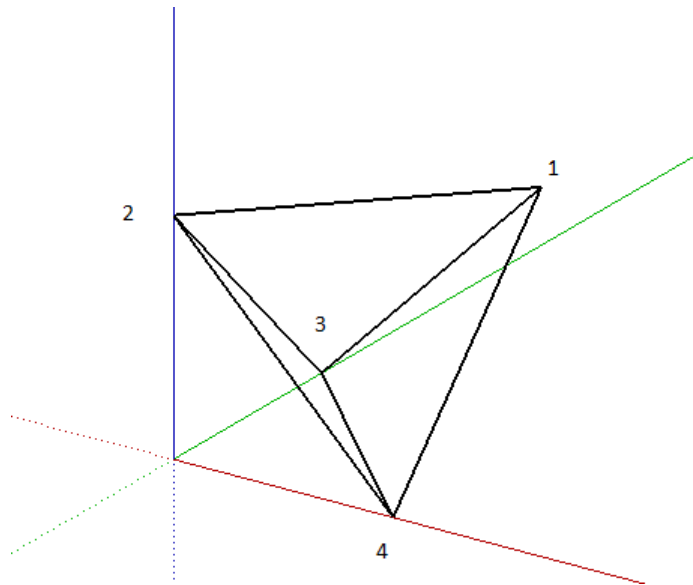
4.2 Cell inflation for a 3-colex

We used Sketchup, a software for drawing in 3D to visualize the construction of a 3-colex using the Cell inflation algorithm. A graph embedded in 3D has four possible cells:

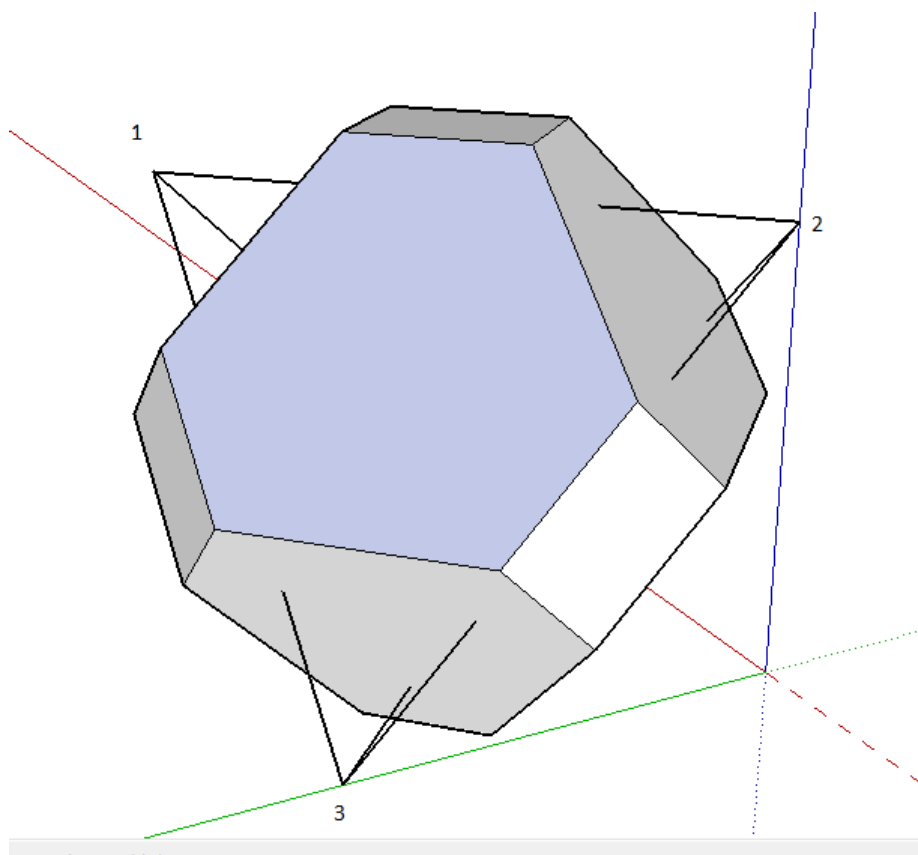
1. 0-cell (Vertices)
2. 1-cell (Edges)
3. 2-cell (Faces)
4. 3-cell (Volumes)

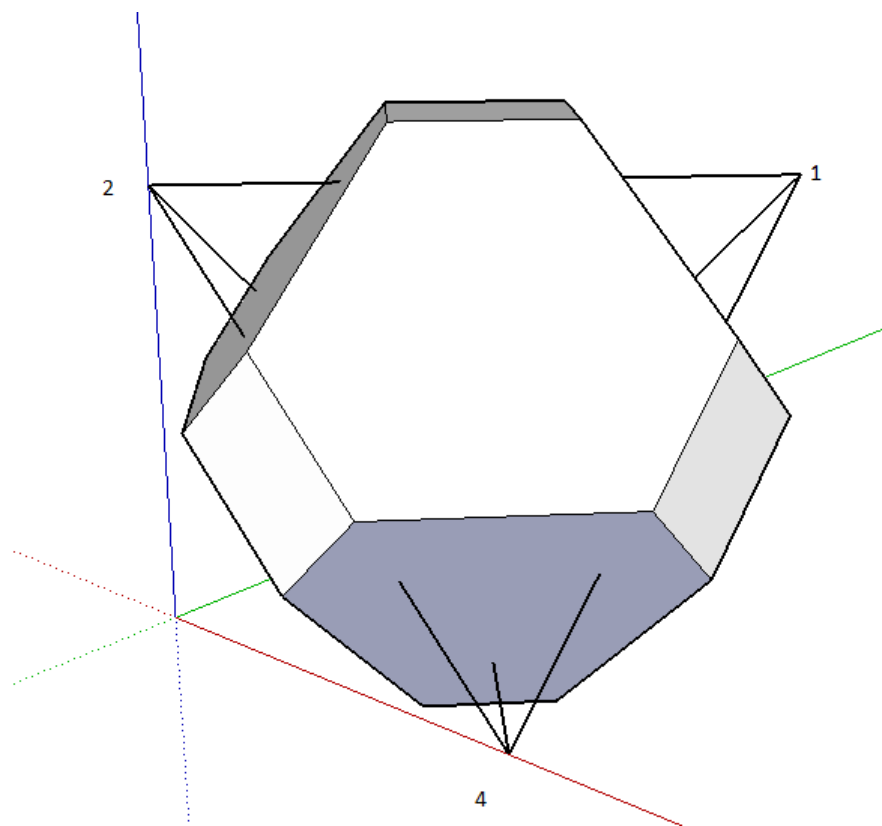
The 3-cell is highest in the hierarchy and extending the Cell inflation algorithm would mean that all the other cells (0,1 and 2) should be blown up into 3-cells. Let us take a simple tetrahedron which has the number of vertices, edges, faces and volumes $s(4, 6, 4, 1)$ - the parameters of the graph. Using sketchup, we can picture the tetrahedron (Note that the colored lines denote the X,Y,Z axes in 3D).



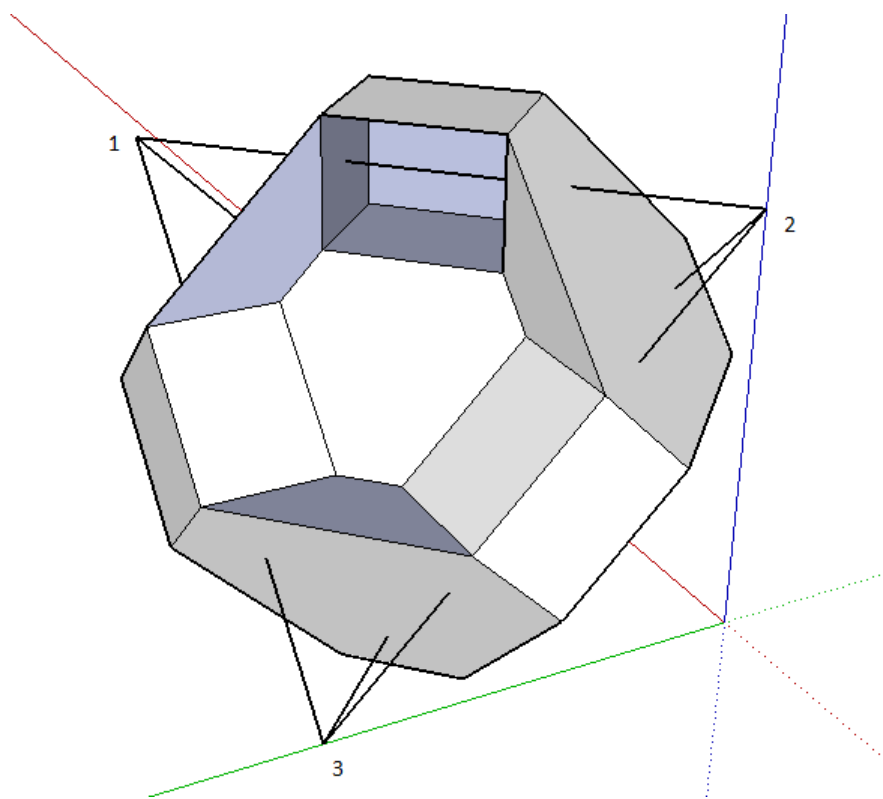


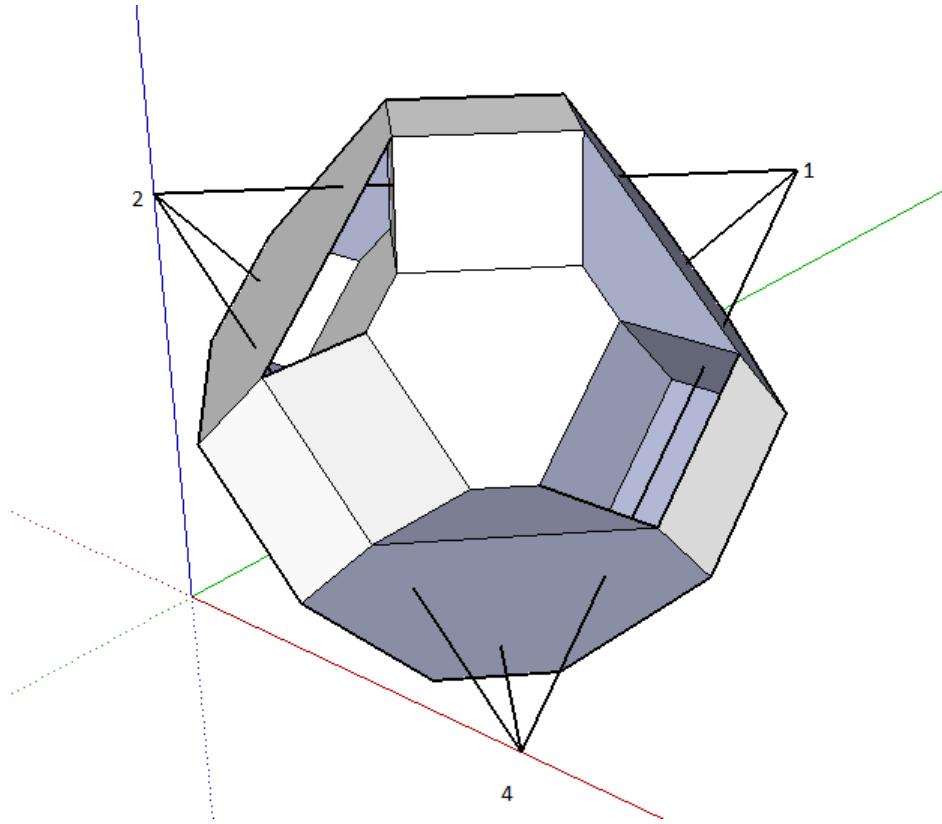
Both the figures represent the same tetrahedron in different views and the vertices are numbered accordingly. Notice that the vertices $\{2, 3, 4\}$ are each on one of the three axes and the vertex 1 is not in any of the axes. We now try and imagine the Cell inflation algorithm for the tetrahedron. After the algorithm is applied, the resultant object looks will be a 3-colex. We picturize this from the same two angles as we did for the tetrahedron.



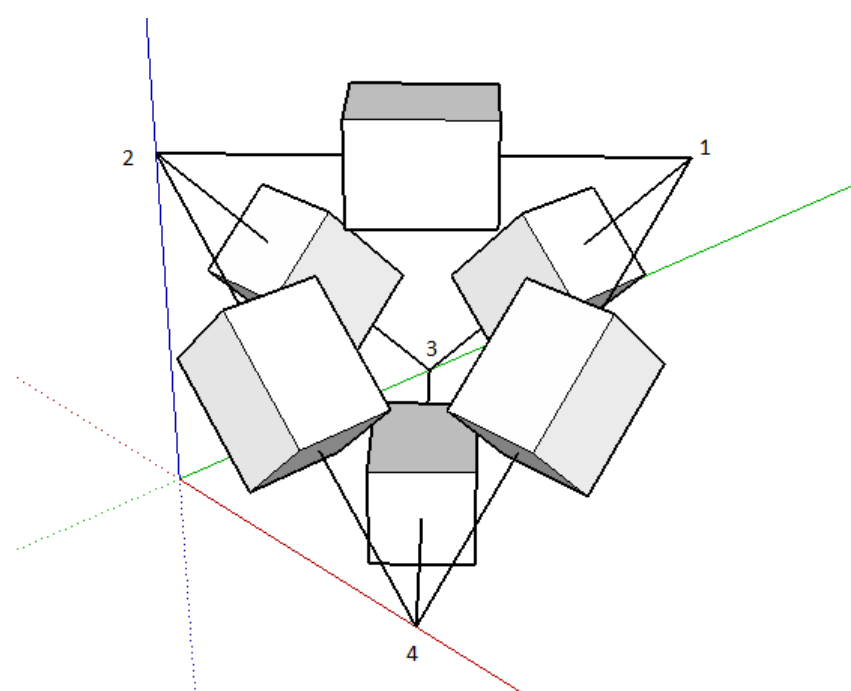
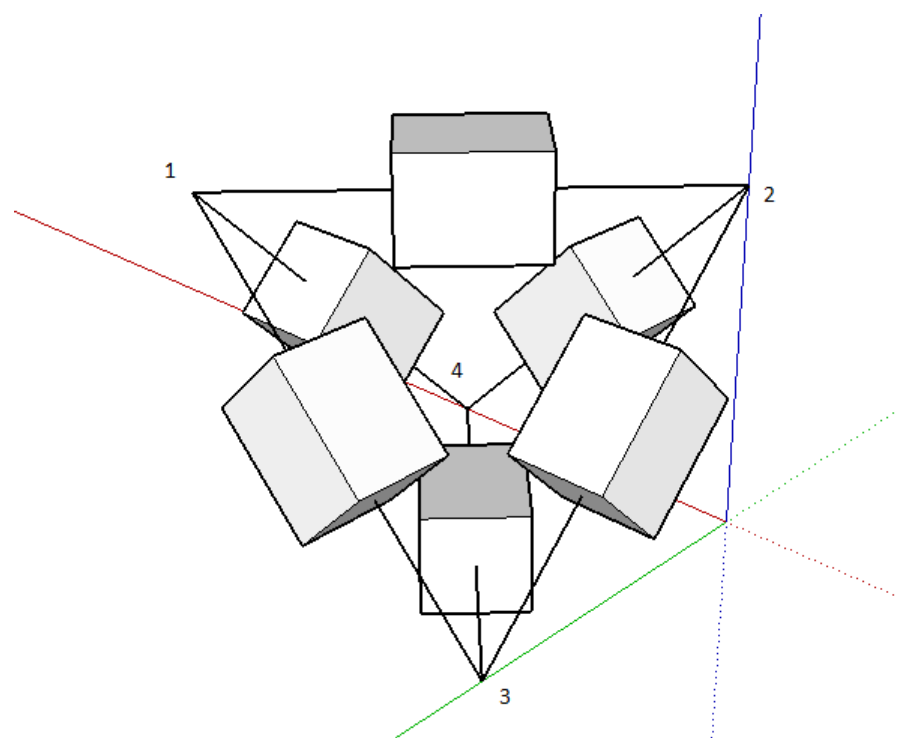


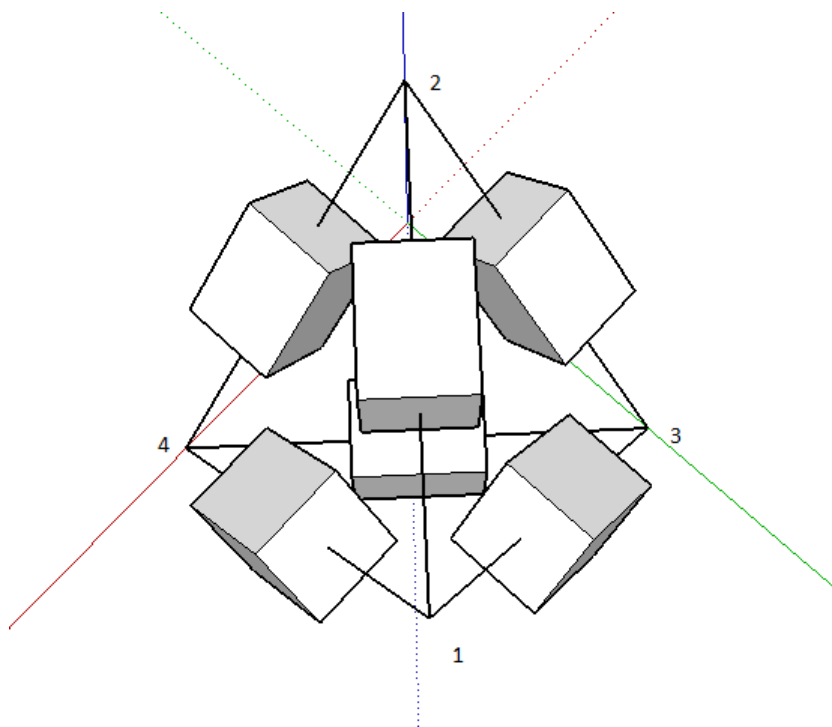
In the last figure, we make some faces transparent so that we can ‘see’ inside the object:



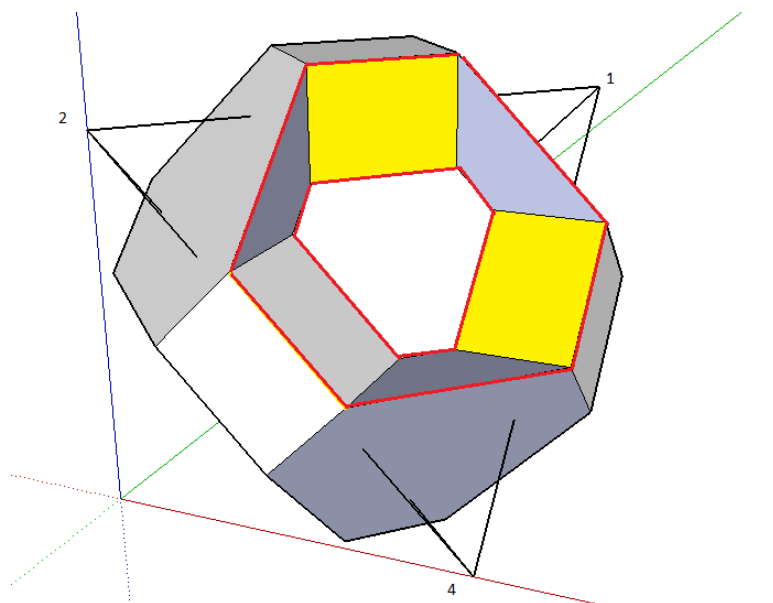


We are interested in limitations that this algorithm induces on the generated colex. We already discussed the same for a 2-colex. Similarly, we wish to perceive this for the colex generated from the tetrahedron. To do this, let us see only the 3-cells that are inflated from 1-cells(edges) of the tetrahedron: And since the stabilizer of the associated color code depends on the facial structure of the 2-colex we see that there are indeed color codes constructed from 2-colexes that cannot be generated from the Cell inflation algorithm.

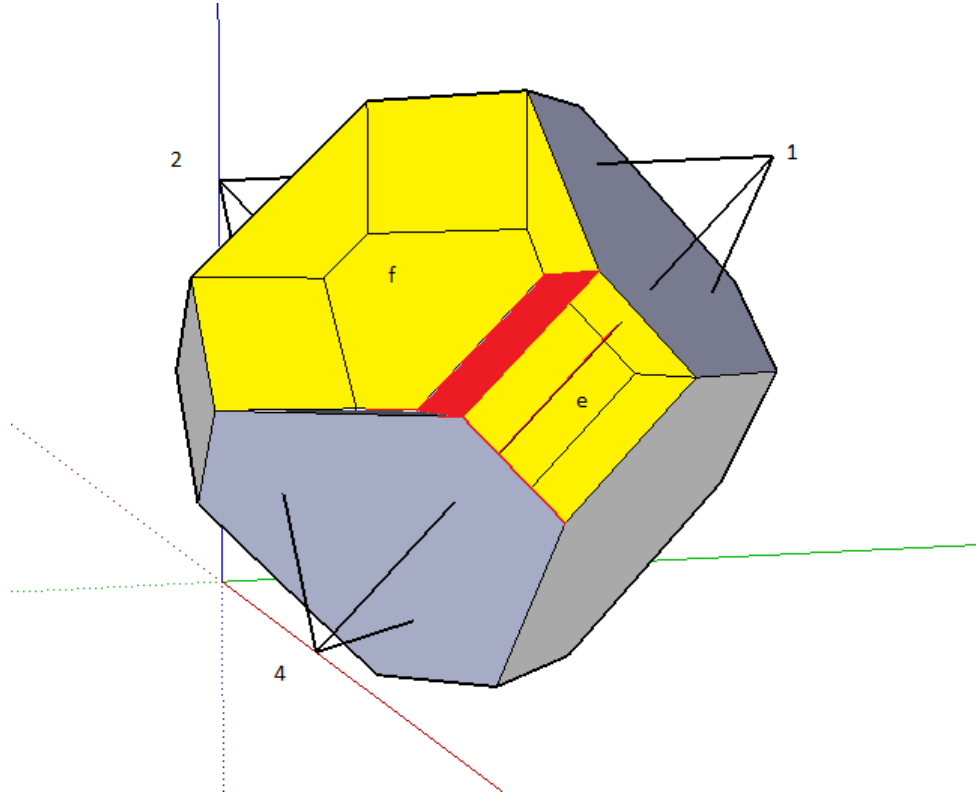




We can clearly see that the volumes(3-cells) derived from edges(1-cells) are in the form of a cuboid. The limitation we observe, however is in the form of a face: The volume derived by inflating a 1-cell and the volume derived from inflating a 2-cell will intersect if the original 1-cell had been in the boundary of the 2-cell. This intersection will be nothing but a face(2-cell is the intersection of two 3-cells). We conjecture that this 2-cell(face) is always four sided irrespective of the graph (embedded in 3D) we start with. Here is a view of this face for the tetrahedron. The red boundary denotes the faces forming the volume derived by inflating a 2-cell(face). We have already seen the volumes from inflating a 1-cell. Their intersection are the faces colored yellow. Notice that they are 4-sided.



In the figure below, e denotes the volume derived from inflating an edge and f is the volume got by inflating a face (both volumes are colored yellow). We can observe that their intersection(the face colored red) is four sided.



Therefore, in the 3D case too, the constructed 3-colex may not be general- the Cell inflation algorithm restricts the set of possible 3-colexes being generated. The stabilizer of the code is heavily dependent on the topology of the colex and hence, the resultant code built from the Cell inflation algorithm is only a restrictive set of color codes made out of 3-colexes when we consider the stabilizer matrix.

We have seen both for 2-colex(with proof) and 3-colex(with an example illustration) that the Cell inflation algorithm may not be the most general way of constructing arbitrary colexes. One suspects it is indeed the case in higher dimensions as well- There is a restriction on some aspect of the colex being generated and hence, this algorithm will not cover all possible color codes built from D-colexes embedded in D dimension. This is where we consider the utility of the Bipartite algorithm.

4.3 Bipartite algorithm for a 2-colex

Unlike the Cell Inflation algorithm, the Bipartite algorithm generates 2-colexes that are far more general. Construction 1 in [2] is the Bipartite algorithm and there is an interesting result proved in [2] that we elaborate here:

Lemma 4.3.1 *Any arbitrary 2-colex can be generated from a bipartite graph embedded in the plane, using the Bipartite algorithm.*

Proof Suppose not. Then there exists some 2-colex (3-valent, 3-face colorable graph embedded on the plane) that cannot be generated from Construction 1 using the Bipartite algorithm. Since the 2-colex is 3-face colorable as well as 3-edge colorable (because of 3-valency) let us color all the faces and edges using three colors, let them be $\{r, g, b\}$. The coloring will be such that if a face is colored with $x \in \{r, g, b\}$ then all the edges on the boundary of the face are colored with $\{r, g, b\} - x$ and any two faces colored with the same color x are linked by an edge that is colored x .

Now, pick a random color, say r and contract *all* edges which are colored differently from r ie, all edges colored b or g . Now because of the coloring and contraction of edges colored b or g , we have faces that are colored r getting shrunk to a point (Because faces colored r have all edges colored b or g). Therefore, our original 3-colorable graph becomes a 2-colorable graph such that faces which were earlier colored r become vertices of the new graph. Taking the dual of such a 2-face colorable graph, we get a bipartite graph (Because only bipartite graphs are 2-colorable). Note that this process is exactly the reverse of the Bipartite construction discussed in 3.2 and hence, the bipartite graph we get now can be the starting graph from which we can implement the Bipartite algorithm to get a 2-colex. ■

What 4.3.1 tells us is that the set of all 2-colexes generated by the Bipartite algorithm will be the set of all possible 2-colexes embedded in the plane! This means all possible color codes that can be obtained from a 2-colex can be generated by starting with a random bipartite graph in the plane and applying the Bipartite construction. [2] also states that given a 2-colex, the bipartite graph from which the algorithm is applied to get that 2-colex need not be unique. Three different bipartite graphs will generate the same 2-colex(hence the same color code).

This means that the Bipartite algorithm is a more general algorithm for generating color codes than the Cell inflation algorithm because the former generates all possible color codes, atleast for 2-Colexes.

4.4 Bipartite algorithm for a 3-colex

We now wonder whether the same analysis is applicable for 3-colexes. We already saw in 4.2 with an illustration that the Cell inflation algorithm will only generate a restricted set of 3-colexes, hence a restricted set of color codes that can be generated from them. Just like what we discussed in 4.3, can we extend the Bipartite construction to 3-colexes that can generate all possible color codes from 3-colexes? It turns out the answer is indeed yes. We are able to apply a similar logic in 3D to get 3-colexes.

Let us call this the **Tripartite algorithm**. We hypothesize the construction of a 3-colex and illustrate the construction with an example In 3.2 we started off with an arbitrary graph embedded in the plane and generated the dual. From the dual, we blew up the 0-cells to get a 2-colex. Therefore, we start here with a *tripartite* graph G embedded in 3D. Note that this graph will have four different cells as

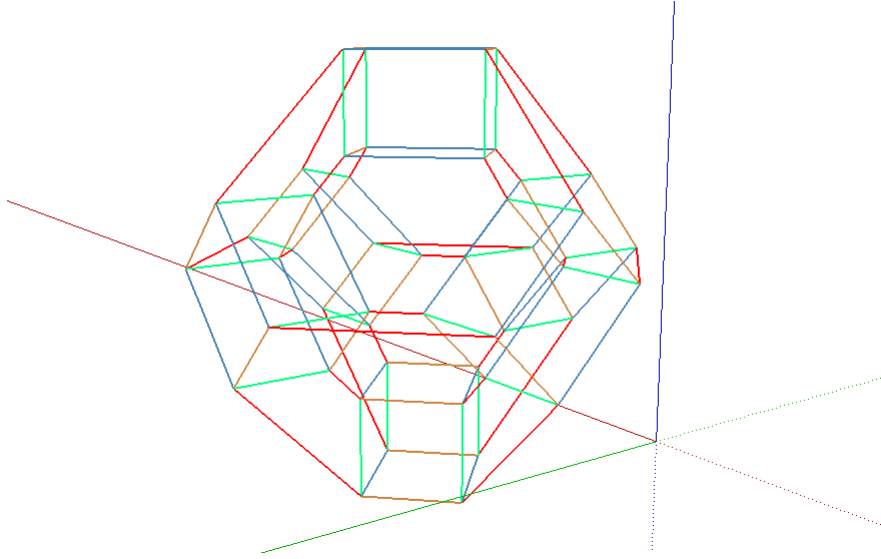
discussed in 4.2. Now we find the dual D of this graph in 3 space.

Dual of a graph in 3D:

1. Any vertex v of G becomes a volume v' of dual D
2. Any volume l of the given graph G becomes a vertex l' of the dual D
3. If in the original graph G , two vertices v_1 and v_2 had an edge e between them, then in the dual D the edge e becomes a face: The volumes v_1 and v_2 will be adjacent with the face e in their boundary.
4. If in G there are two volumes l_1 and l_2 with a face f in common then in the dual D faces become edges: The vertices l_1 and l_2 are adjacent with the edge f in between them.

Therefore, the given graph had (vertices, edges, faces, volumes) to be (v, e, f, l) then the dual graph D will have parameters (l, f, e, v) . Note that because we started with a tripartite graph (which is 3-vertex colorable) the dual graph D will be 3-colorable. But to for a 3-colex we need a 4-valent, 4-face colorable graph. And like the Bipartite algorithm we need to inflate one of the cells.

We hypothesize that the cells which need to be inflated are indeed vertices. We have not provided a proof for this hypothesis but we illustrate the 3-colex using Sketchup. Here is the skeleton of a 3-colex:



Notice the 4-colorability of edges. The same can be done for faces as well, though this is difficult to illustrate.

4.5 Scope for further work

In 4.3.1 we saw that the Bipartite algorithm gives all possible color codes from 2-colexes, unlike the Cell inflation algorithm which has certain limitations in the 2-colex generated as shown in 4.1.1. Is the same applicable for 3-colexes as well?

We have illustrated a possible restriction on the 3-colexes generated by the Cell inflation algorithm in 4.2. We think that the Tripartite algorithm discussed in 4.4 will be general: Any 3-colex can be generated from a tripartite graph in 3D using the Tripartite construction.. There could be a proof similar to the proof for the Bipartite algorithm discussed in 4.3.1.

We are constantly looking forward to moving to higher dimensions for generating color codes from Colexes. Such color codes realize a larger class of fault

tolerant gates and have low complexity in higher dimensions. For this reason, we are interested in generating all color codes from a generic **D-Colex**(A $D + 1$ valent and $D + 1$ colorable graph). We hypothesize that the Cell inflation algorithm always presents a limitation in the D-colexes it produces just like what happens for $D = 2$ and $D = 3$. We also think that to overcome this limitation, the Bipartite/Tripertite algorithms can be extended to higher dimensions as well: ie, any arbitrary D-colex can be generated from a D-partite graph embedded in D-space using a similar algorithm(Finding the dual and inflating some cells) and hence, such a D-partite algorithm will generate all possible color codes from D-colexes.

REFERENCES

- [1] H. Bombin and M. Martin-Delgado, “Exact topological quantum order in $d=3$ and beyond: Branyons and brane-net condensates,” *Phys.Rev.*, vol. B75, p. 075103, 2007.
- [2] P. Sarvepalli and K. R. Brown, “Topological subsystem codes from graphs and hypergraphs,” *Phys. Rev. A*, vol. 86, pp. 23–36, Oct 2012.
- [3] M. A. Nielsen and I. L. Chuang, “Quantum computation and quantum information,” 2013.
- [4] H. Bombin and M. A. Martin-Delgado, “Topological quantum distillation,” *Phys. Rev. Lett.*, vol. 97, p. 180501, Oct 2006.
- [5] H. Bombin and M. A. Martin-Delgado, “Topological computation without braiding,” *Phys. Rev. Lett.*, vol. 98, p. 160502, Apr 2007.
- [6] H. Bombin, “An introduction to topological quantum codes.” ”Topological Codes”, in ”Quantum Error Correction”, edited by Daniel A. Lidar and Todd A. Brun, Cambridge University Press, New York, 2013, 2013.
- [7] D. Gottesman, “An introduction to quantum error correction and fault-tolerant quantum computation,” 2009.
- [8] P. Kaye, R. Laflamme, and M. Mosca, *An Introduction to Quantum Computing*. New York, NY, USA: Oxford University Press, Inc., 2007.

APPENDIX A

Code to generate Voronoi diagram given number of points inside Unit square

Contents

- Vertex information
- Extracting Adjacency matrix
- Finding faces inside $[0,1] \times [0,1]$
- Algorithm for torus looping
- Changing the index of vertices and faces. Very Important Step
- Finding the adjacency Matrix and coordinates matrix to carry over to the algorithm

%Generating random points

```
close all;
poin = 3; % No. of points for starting voronoi tessellation
x = rand(1,poin);
y = rand(1,poin);
xn1 = x-1;
xp1 = x+1;
yn1 = y-1;
yp1 = y+1;
```

```
xn2 = x-2;
xp2 = x+2;
yn2 = y-2;
yp2 = y+2;
```

%Generating Voronoi

```
xnew = [xn2 xn2 xn2 xn2 xn2 xn1 xn1 xn1 xn1 xn1 x x x x xp1
xp1 xp1 xp1 xp1 xp2 xp2 xp2 xp2];
ynew = [yp2 yp1 y yn1 yn2 yp2 yp1 y yn1 yn2 yp2 yp1 y yn1
yn2 yp2 yp1 y yn1 yn2 yp2 yp1 y yn1 yn2];
% figure(1); voronoi(xnew,ynew); %Full voronoi
%
% figure(5);scatter(xnew,ynew,'s');
```

```

% axis([0 1 0 1]);
% figure(2); voronoi(xnew,ynew); %Unit square Voronoi
% axis([0 1 0 1]);
%     box = vert(127,:);
%     box2 = vert(128,:);
%
%     box3 = [box; box2];
%
%     adj2 = [0 1; 1 0];

%Extracting face information using voronoin
z = [xnew' ynew'];
% axis([0 1 0 1]);
[vert,cel] = voronoin(z);
[vx,vy] = voronoi(xnew,ynew);
[ voronoi_face_adjacency ] = get_voronoi_adj(z); %Adjacency (But for faces)
size_voronoi_face_adjacency = size(voronoi_face_adjacency,1);
max_cell = 10; % maximum number of vertices on the boundary of a face, set this number!

size_cel = size(cel,1);
% Number of faces in the Voronoi diagram
face_data = zeros(size_cel,max_cell);
for i=1:size_cel;
    len_cell = size(cel{i},2); % Length of ith row, ie, number of vertices
    %in the boundary of the face i
    for j=len_cell+1:max_cell;
        cel{i} = [cel{i} 0];
    end
    face_data(i,:) = cel{i}; % This contains face information-vertices
    % forming the boundary of each face
end

```

Vertex information

```

size_vert = size(vert,1);
% Number of vertices in the full voronoi diagram

```

Extracting Adjacency matrix

```

full_voronoi_adj = zeros(size_vert); %Adj matrix as zero matrix
for i = 1: size_cel %For every face, we add connectivity information to full_voronoi_adj
    for j = 1:max_cell
        start1 = face_data(i,j);
        if start1 ~= 0
            finish1 = face_data(i,j+1);
            if finish1== 0

```



```

        finish1 = face_data(i,1);
    end
    full_voronoi_adj(start1,finish1) = 1;
    full_voronoi_adj(finish1,start1)=1;
    end
end
end

```

Working on $[0,1] \times [0,1]$

```

    trun_vert = vert; % Considering only vertices in  $[0,1] \times [0,1]$ 
    for i=1:size_vert;
        for j=1:2;
            if (vert(i,j) > 1) || (vert(i,j) < 0);
                trun_vert(i,1) = 0;
                trun_vert(i,2) = 0;
            end
        end
    end
    end

    trun_vert = trun_vert ~= 0; % Logical array for those vertices inside  $[0,1] \times [0,1]$ 
    trun_vert(:,2) = []; % Deletes second column
    ind = find(trun_vert); % This is the list of vertices inside  $[0,1] \times [0,1]$ 

    new_face_data = zeros(size_cel,max_cell);

```

Finding faces inside $[0,1] \times [0,1]$

```

    face_yes = zeros(size_cel,1);
    for i=1:size_cel % Checking if the face has ANY point in  $[0,1] \times [0,1]$ 
        for j =1:max_cell
            temp3 = face_data(i,j);
            temp4 = ismember(temp3,ind);
            if temp4 ==1
                face_yes(i) = 1;
                % Those faces which have atleast 1 vertex inside  $[0,1] \times [0,1]$ 
                new_face_data(i,:) = face_data(i,:);
            end
        end
    end

    end % Now we only need to check these faces where face_yes =1
    faces_list = find(face_yes); %List of faces inside  $[0,1] \times [0,1]$ 

```

Algorithm for torus looping

```

    for i= 1:size_cel % 1:size_cel faces on  $[0,1] \times [0,1]$  with torus embedding
        if face_yes(i) ==1

```

```

for j =1:max_cell % 1:max_cell
temp_venum = new_face_data(i,j); %Current vertex
if temp_venum ~= 0;
temp_nextve = face_data(i,j+1); %If current vertex non zero,
%get the next vertex
temp5 = ismember(temp_venum,ind); %Is current vertex in [0,1]x[0,1]
if temp5 ==0; %If current vertex is not in the list, make it zero
new_face_data(i,j)=0;
end

if temp_nextve ==0; %Next vertex is zero
temp_nextve= face_data(i,1); %Move to beginning of the row
temp6 = ismember(temp_nextve,ind); %Is nextvertex in the list

if temp6 ==0; %If nextvertex is zero & not in the list,we need to
%find prop vertex in list
t1 = vert(temp_nextve,1);
t2 = vert(temp_nextve,2);
if(t1 <0 && t2<0);
t3 = t1+1;
t4= t2+1;
row1 = [t3 t4];
temp_new = find(ismemberf(vert,row1,'rows','tol',0.001));
new_face_data(i,1)= temp_new;
end
if(t1 <0 && 0<t2 && t2<1);
t3 = t1+1;
t4= t2;
row1 = [t3 t4];
temp_new = find(ismemberf(vert,row1,'rows','tol',0.001));
new_face_data(i,1)= temp_new;
end
if(t1 <0 && t2>1);
t3 = t1+1;
t4= t2-1;
row1 = [t3 t4];
temp_new = find(ismemberf(vert,row1,'rows','tol',0.001));
new_face_data(i,1)= temp_new;
end
if(0<t1 && t1 <1 && t2<0);
t3 = t1;
t4= t2+1;
row1 = [t3 t4];
temp_new = find(ismemberf(vert,row1,'rows','tol',0.001));
new_face_data(i,1)= temp_new;
end
end

```

```

        if(0<t1 && t1 <1 && 0<t2 && t2<1);
            t3 = t1;
            t4= t2;
            row1 = [t3 t4];
            temp_new = find(ismemberf(vert,row1,'rows','tol',0.001));
            new_face_data(i,1)= temp_new;
        end
        if(0<t1 &&t1 <1 && t2>1);
            t3 = t1;
            t4= t2-1;
            row1 = [t3 t4];
            temp_new = find(ismemberf(vert,row1,'rows','tol',0.001));
            new_face_data(i,1)= temp_new;
        end
        if(t1>1 && t2<0);
            t3 = t1-1;
            t4= t2+1;
            row1 = [t3 t4];
            temp_new = find(ismemberf(vert,row1,'rows','tol',0.001));
            new_face_data(i,1)= temp_new;
        end
        if(t1>1 && 0<t2 &&t2<1);
            t3 = t1-1;
            t4= t2;
            row1 = [t3 t4];
            temp_new = find(ismemberf(vert,row1,'rows','tol',0.001));
            new_face_data(i,1)= temp_new;
        end
        if(t1>1 && t2>1);
            t3 = t1-1;
            t4= t2-1;
            row1 = [t3 t4];
            temp_new = find(ismemberf(vert,row1,'rows','tol',0.001));
            new_face_data(i,j)= temp_new;
        end
    end

else %If next vertex is non zero
    temp6 = ismember(temp_nextve,ind);
    if temp6 ==0; %If nextvertex is non zero and not in the list
        t1 = vert(temp_nextve,1);
        t2 = vert(temp_nextve,2);
        if(t1 <0 && t2<0);
            t3 = t1+1;
            t4= t2+1;
            row1 = [t3 t4];

```

```

        temp_new = find(ismemberf(ver, row1, 'rows', 'tol', 0.001));
        new_face_data(i, j+1) = temp_new;
    end
    if(t1 < 0 && 0 < t2 && t2 < 1)
        t3 = t1+1;
        t4 = t2;
        row1 = [t3 t4];
        temp_new = find(ismemberf(ver, row1, 'rows', 'tol', 0.001));
        new_face_data(i, j+1) = temp_new;
    end
    if(t1 < 0 && t2 > 1);
        t3 = t1+1;
        t4 = t2-1;
        row1 = [t3 t4];
        temp_new = find(ismemberf(ver, row1, 'rows', 'tol', 0.001));
        new_face_data(i, j+1) = temp_new;
    end
    if(0 < t1 && t1 < 1 && t2 < 0);
        t3 = t1;
        t4 = t2+1;
        row1 = [t3 t4];
        temp_new = find(ismemberf(ver, row1, 'rows', 'tol', 0.001));
        new_face_data(i, j+1) = temp_new;
    end
    if(0 < t1 && t1 < 1 && 0 < t2 && t2 < 1);
        t3 = t1;
        t4 = t2;
        row1 = [t3 t4];
        temp_new = find(ismemberf(ver, row1, 'rows', 'tol', 0.001));
        new_face_data(i, j+1) = temp_new;
    end
    if(0 < t1 && t1 < 1 && t2 > 1);
        t3 = t1;
        t4 = t2-1;
        row1 = [t3 t4];
        temp_new = find(ismemberf(ver, row1, 'rows', 'tol', 0.001));
        new_face_data(i, j+1) = temp_new;
    end
    if(t1 > 1 && t2 < 0);
        t3 = t1-1;
        t4 = t2+1;
        row1 = [t3 t4];
        temp_new = find(ismemberf(ver, row1, 'rows', 'tol', 0.001));
        new_face_data(i, j+1) = temp_new;
    end
    if(t1 > 1 && 0 < t2 && t2 < 1);

```



```

%This contains new vertex numbering and old label, along with coordinates

modi_new_face_data(modi_new_face_data==temp2) = i;
% New face and vertex labelled matrix
end

```

Finding the adjacency Matrix and coordinates matrix to carry over to the algorithm

```

voronoi_adj = zeros(size_trun_vert); %Adj matrix as zero matrix
for i = 1: size_faces_list %For every face, we add connectivity information to voronoi_adj
    for j = 1:max_cell
        start = modi_new_face_data(i,j);
        if start ~= 0
            finish = modi_new_face_data(i,j+1);
            if finish== 0
                finish = modi_new_face_data(i,1);
            end
            voronoi_adj(start,finish) = 1;
            voronoi_adj(finish,start)=1;
        end
    end
end

```

```

voronoi_adj; %ADJACENCY MATRIX
vertices_matrix = modi_trun_vert(:,1:2); % VERTICES MATRIX

```

```

box = vert(127,:);
box2 = vert(128,:);
box3 = vert(109,:);
box4 = vert(15,:);

```

```

box5 = [box; box3;box4;box2];

```

```

adj2 = [ 0 1 0 0; 1 0 0 0; 0 0 0 1; 0 0 1 0]
figure(1); voronoi(xnew,ynew); %Full voronoi

```

```

figure(5);scatter(xnew,ynew,'s');
axis([0 1 0 1]);
figure(2); voronoi(xnew,ynew); %Unit square Voronoi
axis([0 1 0 1]);
hold on;

```

APPENDIX B

This code runs the Bombin(Cell inflation algorithm) given the lattice embedded in a torus and its information

Contents

- Generating initial graph
- Running Bombin's algorithm
- Joining the lines
- Sanity check

Generating initial graph

```
n=size(voronoi_adj,1) coords=vert(1:n,:); coords = rand(n,2); gplot(T, coords,
'-.*') coords = [ 5 3; 3 4; 4 6; 6 7.5 ; 7 6; 7.5 4; 7 3];
```

```
T = full_voronoi_adj;
n= size(T,1);
coords = vert;
% coords=rand(n,2);
working = T;
coordscopy=coords;
coordsinfo = [coordscopy zeros(n,2)];
% n=7;
```

Running Bombin's algorithm

```
ln = n; %number of ending nodes, this will be updated
curr=1; %Current node
startlen=n; %Total number of starting nodes
theta = pi/12; %Angle of rotation
```

```
dist_mat = zeros(startlen,startlen); %This loop finds the vertex of
%(and)minimum distance for a given vertex
dist_min = zeros(startlen,2);
```

```

for curr= 1:startlen;
    for i = 1:startlen;
        if working(curr,i) ~=0
            x1 = coords(curr,1);
            y1 = coords(curr,2);
            x2 = coords(i,1);
            y2 = coords(i,2);
            i;
            dist_mat(curr,i) = sqrt( (x1-x2)^2 + (y1-y2)^2 );
        end
    end
end

temp8 = dist_mat(curr,:);
temp8(~temp8) = nan;
[dist_min(curr,1),dist_min(curr,2)] = min(temp8); % Minimum non zero
%entry of the row 'curr'
dist_min(curr,2) = uint8(dist_min(curr,2));
%Index of vertex for which this minimum occurs
if isnan(dist_min(curr,1))
    dist_min(curr,2) =0;
end
end

end

bignum = 20; %Careful while scaling up!
vertex_info = zeros(startlen);
currnum =1;

for curr=1:startlen;
    for i = curr+1:startlen; %So that we work only with upper triangular
        %part of the adjacency matrix
        if working(curr, i)~=0;
            %Adds new node if curr and i connected
            working=[working zeros(ln,2)];
            vertex_info(curr,i) =ln+1;
            ln=ln+2;
            working=[working; zeros(2, ln)];

            ln2 = ln-1; % for left new node
            ln3 = ln;   % for right new node

            working(ln2,i)=1;
            %Add new edge between created LEFT new node and i
            working(i,ln2)=1;

            working(ln3,i)=1;
            %Add new edge between created RIGHT new node and i

```



```

working(i,ln3)=1;

working(ln2,ln3)=1; %Connect the two newly created vertices
working(ln3,ln2)=1;

%Calc coords of first new point
temp_9 = dist_min(curr,1);
temp_10=dist_min(curr,2);
x1 = coords(curr,1);
y1 = coords(curr,2);
x2 = coords(temp_10,1);
y2 = coords(temp_10,2);
x3 = coords(i,1);
y3 = coords(i,2);

temp_dist = sqrt( (x1-x3)^2 + (y1-y3)^2 );
%Distance between curr and i
temp_d_min = sqrt( (x1-x2)^2 + (y1-y2)^2 );
% Min distance among all i's connected to curr

lamda = 0.25*(temp_d_min)/(temp_dist); %Ratio to be considered
newcoords=(1-lamda)*coords(curr, :)+lamda*coords(i,:);
%Point on the line connecting curr and i
x4=newcoords(1);
y4=newcoords(2);
if isnan(dist_min(curr,1))
    dist_min(curr,2) = 0;
    % vertex 'curr' has no edges incident on it
end
if dist_min(curr,2) ~= 0;
    %Rotation by theta of the point on the line about curr
    a = [x1 y1];
    b = [x4,y4];
    c = b-a;
    rot = [cos(theta) -sin(theta);sin(theta) cos(theta)];
    %left point
    rot1= [cos(theta) sin(theta); -sin(theta) cos(theta)];
    %right point
    d = c*rot + a;
    e = c*(rot1) +a;

end

coords=[coords;d;e];
coordsinfo=[coordsinfo;d curr i;e curr i];

```

```

        working(curr, i)=0;%Removing the original i-curr connection
        working(i, curr)=0;
    end
end
end
%
%   figure(1); gplot(T, coords, '-*')
%   figure(2); gplot(working, coords, '-*')

%New loop. Takes care of Introduced edges( post-split)
lengthloop=ln;
for curr= 2:startlen;
    for i=startlen+1:lengthloop;
        if working(curr, i)~=0

            working=[working zeros(ln,2)];
            orig_vert = coordsinfo(i,3);
            %original vertex from which i was generated
            vertex_info(curr,orig_vert) = ln+1;
            ln=ln+2;
            working=[working; zeros(2, ln)];

            ln2 = ln-1; % for a-clockwise new node
            ln3 = ln;   % for clockwise new node

            %           working(ln2,i)=1;
            %Add new edge between created A-clockwise new node and i
            %           working(i,ln2)=1;
            %
            %           working(ln3,i)=1;
            %Add new edge between created clockwise new node and i
            %           working(i,ln3)=1;

            %Calc coords of first new point
            temp_9 = dist_min(curr,1);
            temp_10=dist_min(curr,2);
            x1 = coords(curr,1);
            y1 = coords(curr,2);
            x2 = coords(temp_10,1);
            y2 = coords(temp_10,2);
            orig_vert = coordsinfo(i,3);
            %original vertex from which i was generated
            x3 = coords(orig_vert,1);
            y3 = coords(orig_vert,2);

            temp_dist = sqrt( (x1-x3)^2 + (y1-y3)^2 );

```

```

%Distance between curr and i
temp_d_min = sqrt( (x1-x2)^2 + (y1-y2)^2 );
% Min distance among all i's connected to curr

lamda = 0.25* temp_d_min/temp_dist; %Ratio to be considered
newcoords=(1-lamda)*coords(curr, :)+ lamda*coords(orig_vert,:);
%Point on the line connecting curr and i
x4=newcoords(1);
y4=newcoords(2);
if isnan(dist_min(curr,1))
    dist_min(curr,2) = 0;
    % vertex 'curr' has no edges incident on it
end
if dist_min(curr,2) ~= 0;
    %Rotation by theta of the point on the line about curr
    a = [x1 y1];
    b = [x4,y4];
    c = b-a;
    rot = [cos(theta) -sin(theta);sin(theta) cos(theta)];
    %a-clockwise point
    rot1= [cos(theta) sin(theta); -sin(theta) cos(theta)];
    %clockwise point
    d = c*rot + a;
    e = c*(rot1) +a;

end

coords=[coords;d;e];
coordsinfo=[coordsinfo;d curr orig_vert;e curr orig_vert];

working(curr, i)=0;
%Removing the original i-curr connection
working(i, curr)=0;

working(curr,i+1)=0;
%Removing the original i+1-curr connection
working(i+1,curr)=0;

% Distance comparison- To get a square from an expanded edge
dista=coords(ln2,:);
distb=coords(ln3,:);

distc=coords(i,:);
distd=coords(i+1,:);

distac=norm(distc-dista);

```

```

        distad=norm(distd-dista);
        distbc=norm(distb-distc);
        distbd=norm(distb-distd);

        if (distac+distbd) > (distad+distbc)
            working(ln2, i+1)=1;
            working(ln3, i)=1;

            working(i+1, ln2)=1;
            working(i, ln3)=1;

        else
            working(ln2, i)=1;
            working(ln3, i+1)=1;

            working(i, ln2)=1;
            working(i+1, ln3)=1;
        end

        working(ln2, ln3)=1;
        working(ln3, ln2)=1;

        working(curr, i)=0;
        working(curr, i+1)=0;

        working(i, curr)=0;
        working(i+1, curr)=0;

        i=i+2;

    end
end
end

```

Joining the lines

```

%Converts face_data into clockwise order
for i = 1:size_cel
    nonzer = find(face_data(i,:));
    nonzer_len = size(nonzer,2);
    x_row = zeros(1,nonzer_len +1);
    y_row = x_row;

    for j=1:nonzer_len
        temp_index = face_data(i,j);
    end
end

```

```

        x_row(1,j) = vert(temp_index,1);
        y_row(1,j) = vert(temp_index,2);
    end
    x_row(1,nonzer_len+1) = vert(face_data(i,1),1);
    y_row(1,nonzer_len+1) = vert(face_data(i,1),2);

    x6 = x_row;
    y6 = y_row;

    clk = ispolycw(x_row,y_row); %are the vertices listed in clockwise order

    if clk ==0
        temp_array2 = face_data(i,1:nonzer_len);
        for k =1:nonzer_len
            face_data(i,nonzer_len+1-k) = temp_array2(k);
        end

        [x_row y_row] = poly2cw(x6,y6);
    end

    clk2 = ispolycw(x_row,y_row);

end

%Now, to join the lines

for i= 1:size_cel %1:size_cel %Doing it face by face
    nonzer = find(face_data(i,:)); % List of vertices in face i
    nonzer_len = size(nonzer,2); %Number of vertices in face i
    per_face_mat = zeros(1,2*nonzer_len);
    for j = 1:nonzer_len %1:nonzer_len %Going vertex by vertex
        vert1 = face_data(i,j); %Curr vertex
        vert2 = face_data(i,j+1);
        if j == nonzer_len
            vert2 = face_data(i,1);
        end

        temp13 = vertex_info(vert1,vert2);
        temp14 = vertex_info(vert2,vert1)+1;

        %
        % low = min([vert1 vert2]); %Lower among the two.
        %
        %
        % if low == vert1
        %     temp13 = vertex_info(vert1,vert2)+1;
        %     temp14 = vertex_info(vert2,vert1);
        % else

```

```

%           temp13 = vertex_info(vert1,vert2);
%           temp14 = vertex_info(vert2,vert1)+1;
%           end

           q = 2*j-1;
           per_face_mat(1,q) = temp13;
           per_face_mat(1,q+1) = temp14; %creating sequence matrix of vertices
           % lying inside the given face
       end

       for m = 1:2*nonzer_len % 1:2*nonzer_len
           temp15 = per_face_mat(1,m);
           if m == 2*nonzer_len
               temp16 = per_face_mat(1,1);
           else
               temp16 = per_face_mat(1,m+1);
           end
           working(temp15,temp16) =1;
           working(temp16,temp15) =1;
       end
   end
   % per_face_mat

%
%           if j==1
%               vert3= face_data(i,nonzer_len);
%           else
%               vert3 = face_data(i,j-1);
%           end
%           after2 = vertex_info(vert1,vert2)+1; %Point created by splitting j-j+1 edge
%           before2 = vertex_info(vert1,vert3)+1; %Point created by splitting j- j-1 edge
%
%           working(after2,before2) =1;
%           working(before2,after2) =1;

%%Plotting the figures
figure(3); gplot(T, coordscopy, '-*')
axis([0 1 0 1]);
figure(4); gplot(working, coords, '-*')
axis([0 1 0 1]);

```

```
% text( working(1,1), figure(1,2), 'Label first node');
```

Sanity check

```
a = size(working,1) - n;  
b = nnz(T)/2;
```

```
a - 4*b %Condition for 2-colex: this should be zero
```

APPENDIX C

Code to implement Bipartite algorithm for a given bipartite graph embedded in the plane

Contents

- INPUTS : SQUARE LATTICE in $[0,m] \times [0,n]$
- Inputs- generic case
- Finding the dual of G
- Blowing up the vertices
- Getting the stabilizer matrix
- Plotting the graphs

%Construction 1 implementation- Getting a 2-colex from arbitrary embedded bipartite graph

function Colex_bipartite(clen,rlen)

% Note: We perform this here for special cases, for example square lattice, square octogonal lattice etc. which are obviously bipartite.

%Assumption: The graph is given to us in the form of adjacency matrix(vertex and edge sets information) and face information

% Abbreviations: Given graph: G, Dual graph: D

INPUTS : SQUARE LATTICE in $[0,m] \times [0,n]$

close all;

% clen = 3; % clen is the number of cells per row

% rlen = 5; % rlen is the number of cells per column

m=1; % Temp variable

num_vert = (clen+1)*(rlen+1); % Number of vertices of G

vert_set = zeros(num_vert,2); % Set of vertices of G

adj_mat = zeros(num_vert,num_vert); %Adjacency matrix of G

face_info = zeros(clen*rlen,4); % Face info matrix, each face has 4 vertices,i.e the column size


```

% Getting the vertex set
for i=0:rlen
    for j=0:clen
        vert_set(m,:) = [j i]; %Coordinates of the vertex set of G
        m=m+1; % Moving to next vertex indexing
    end
end

%Getting the Adjacency matrix
for i=0:rlen
    n= (clen+1)*i; % Temp variable
    for j=1:clen
        adj_mat(n+j,n+j+1) = 1;
        adj_mat(n+j+1,n+j) = 1;    %Connecting horizontal lines
    end
end

for j=0:rlen-1
    n = (clen+1)*j; %temp variable
    for i=1:clen+1
        adj_mat(n+i,n+i+(clen+1)) = 1;
        adj_mat(n+i+(clen+1),n+i) = 1;    %Connecting vertical lines
    end
end

%Getting the face data matrix
m=1; %Temp variable, index of the faces
for j=0:rlen-1
    for i = 1+(clen+1)*j:clen+(clen+1)*j
        face_info(m,:) = [i i+1 i+1+clen+1 i+clen+1];
        %Storing the vertex sequence of face m from bottom left to top left
        %counterclockwise
        m=m+1; %Next face indexing
    end
end

Error using Colex_bipartite (line 20)
Not enough input arguments.

```

Inputs- generic case

%In general, we may only be given the below three matrices, however we have

%derived them for a square lattice.

%Parameters of the given graph G
vert_set; %Vertex coordinates and numbering of G
adj_mat; %Adjacency matrix of G, with size = size(vert_set)
face_info; % Face information of G. f faces counted anticlockwise from
% bottom left to top left

v = size(vert_set,1); % Number of vertices of G
f = size(face_info,1); % Number of faces of G excluding infinite face

Finding the dual of G

%Finding the vertices of the Dual D

dual_vert = zeros(f,2); %Because if G had f faces, the dual D has f vertices

for i=1:f
temp = face_info(i,:); % temp is the face i of G
nzer = temp(temp~=0); %Lists only non zero vertices in temp, ie actual vertices
size_nzer = size(nzer,2); % Number of vertices in face i of G.

%To find the centroid of the points on the face- The centroid of face
%in G will be the vertex of the dual graph D. Note that we will take
%care of the infinite face next, not on this routine.

%Formula to find centroid of closed polygon with vertices not
%intersecting is The centroid of a non-self-intersecting closed polygon
%is known and available. The following loop will perform that

nzer = [nzer nzer(1)];
% making the face order a cycle by putting the first vertex in the last also

c_x = 0;
c_y = 0;
area_face = 0;
for j =1:size_nzer
x_j = vert_set(nzer(j),1); % x coordinate of jth vertex of face i
x_jplus1 = vert_set(nzer(j+1),1); % x coordinate of (j+1)th vertex of face i
y_j = vert_set(nzer(j),2); % y coordinate of jth vertex of face i
y_jplus1 = vert_set(nzer(j+1),2); % y coordinate of (j+1)th vertex of face i

c_x = c_x + (x_j + x_jplus1)*(x_j * y_jplus1 - x_jplus1 * y_j);
% This is the scaled version of x coordinate of centroid

```

        c_y = c_y + (y_j + y_jplus1) * (x_j * y_jplus1 - x_jplus1 * y_j);
        % This is the scaled version of x coordinate of centroid

        area_face = area_face + 0.5 * (x_j * y_jplus1 - x_jplus1 * y_j);
        % Finds the area of the face

    end

    dual_vert(i,:) = (1/6*area_face) * [c_x c_y]; % Final step to find centroid-
    %Divide c_x and c_y by 6*Area of centroid
end

dual_vert; %This matrix contains the coordinates of vertices of D, size f,
%except the infinite face.

% We will now deal with the infinite face.

% For every edge of a face adjacent to the infinite face,
%we join the centroid of the face with the
% midpoint of that edge, say this distance is d and extend the line by
% another d to get a point on the infinite face. This point in the dual
% will be the vertex of the dual corresponding to the infinite face in the
% original graph. Note that we do this to all faces bordering infinite face
% and that now, all these new vertices created are in the SAME, ie, they
% only denote one vertex

m=1;
dual_extvert = [];
for i = 1 :v
    temp31 = ismember(face_info,i);
    [irows3,icols3] = find(temp31);

    for j = i+1:v
        temp32 = ismember(face_info,j);
        [irows4,icols4] = find(temp32);

        temp33 = intersect(irows3,irows4);

        if nnz(temp33) ==1

            len_temp33 = nnz(face_info(temp33,:));
            temp34 = face_info(temp33,1:len_temp33);

            temp41 = find(ismember(temp34,i));

```

```

temp42 = find(ismember(temp34,j));

temp43 = abs(temp41-temp42);

if (temp43 ==1) || (temp43 == len_temp33-1)
    first_vert = i;
    second_vert = j;

    vert1 = vert_set(first_vert,:);
    vert2 = vert_set(second_vert,:);

    midvert = 0.5*(vert1) + 0.5*(vert2);

    temp35 = dual_vert(temp33,:);

    dist1 = pdist([midvert;temp35]);

    temp36 = 2*(midvert) - temp35;
    temp37 = [m temp33 i j temp36];

    dual_extvert(m,:) = temp37;
    m=m+1;
end

end

end

end

%Now, the entire vertex set of the dual consists of faces of the original
%graph and the external vertices that we calculated, but not that all these
%vertices denote only one vertex- that due to the infinite face.

dual_vert_all = [dual_vert; dual_extvert(:,[5,6])];
%This will be the full set of vertices.

% Finding the Adjacency matrix of Dual D

dual_adj = zeros(f,f); %Adjacency matrix of dual D, size f x f.

for i=1:v
    temp4 = ismember(face_info,i); % Returns a 1 in all faces where i is present in G
    [irow,icol] = find(temp4); %irow will give all the faces where i occurs in G
    for j =1:v
        temp5 = ismember(face_info,j); % Returns a 1 in all faces where j is present in G

```

```

[jrow,jcol] = find(temp5); %Finds the rows and columns of all such faces in G

if i~=j
    common_row = intersect(irow,jrow); % Returns the rows where
    % i and j both present

    % If common_row were empty, then it means the edge i-j is not a
    % boundary of two faces. If common_row were non empty, it will
    % return a 2x1 matrix with both the common rows(which are the
    % common faces) of the edge i-j. However, for edges that belong
    % to exactly one face (ie, the other face is the infinite face)
    % common_row has 1 element,we will weed out this too

    empty_check = isempty(common_row); % Checks if common_row is empty
    size_common_row = size(common_row,1);

    if empty_check ==0 && size_common_row ==2 % if i-j were a boundary of
    % two faces of G
        temp5 = common_row(1); % one of the faces of G containing i-j
        temp6 = common_row(2); % other face of G containing i-j
        dual_adj(temp5,temp6) =1;
        dual_adj(temp6,temp5)=1; %In the dual D, these two vertices are adjacent
    end
end
end
end

dual_adj; %Adjacency matrix of the dual graph of size f x f without external vertices.

%we will find th adjacency matrix with external vertices

dual_adj_full = zeros((f+size(dual_extvert,1)),(f+size(dual_extvert,1)));
dual_adj_full(1:f,1:f) = dual_adj;

for i=1:f
    temp44 = ismember(dual_extvert(:,[2]),i);
    temp45 = find(temp44);

    if isempty(temp45) == 0
        len_temp45 = size(temp45,1);

        for j=1:len_temp45
            dual_adj_full(i,f+temp45(j))=1;
            dual_adj_full(f+temp45(j),i)=1;
        end
    end
end

```

```

    end
end

% Finding the face ordering of dual D

size_dual_faces = zeros(v,1);
%This part is to find out the maximum number of vertices a face in the dual D can have
for i = 1:v
    temp7 = ismember(face_info,i); % Returns a 1 in all faces where i is present in G
    [irow,icol] = find(temp7); %irow will give all the faces where i occurs in G

    size_dual_faces(i) = size(irow,1);
    % Counts the number of faces of G having vertex i .
    % This will be equal to the number of edges in face i of the dual D.
end
Maxi = max(size_dual_faces);
% Maxi is the maximum possible number of vertices a face of D can have

dual_face_info = zeros(v,Maxi); %face info of dual D, v faces are there.

for i = 1:v
    temp7 = ismember(face_info,i); % Returns a 1 in all faces where i is present in G
    [irow,icol] = find(temp7); %irow will give all the faces where i occurs in G

    face_count = size(irow,1); % Counts the number of faces of G having vertex i .
    % This will be equal to the number of edges in boundary of face i in the dual D.
    for j=1:face_count
        dual_face_info(i,j) = irow(j,1); %Inputting vertices in a-clockwise for
        % face info in D
    end
end

% parameters of the dual graph D
dual_vert; %This matrix contains the coordinates of vertices of D, size f
dual_adj; %Adjacency matrix of the dual graph of size f x f
dual_face_info; %Face information of the dual graph, size v x Maxi, a-clockwise

v_dual = size(dual_vert,1); % Number of vertices of dual graph
f_dual = size(dual_face_info,1); % number of faces of the dual graph

```

Blowing up the vertices

```

dual_num_edge = nnz(dual_adj); % twice the number of edges in the dual graph

```

```

colex_vert = zeros(dual_num_edge,4); % Vertices for the colex
colex_adj = zeros(dual_num_edge,dual_num_edge); % Adjacency for the colex
colex_face_info = []; % This will be the face matrix of the colex

% Note, we are ignoring the infinite face here.
colex_adj=zeros(dual_num_edge,dual_num_edge); %Adjacency matrix of the colex
m=1;

upd_dual_face_info = []; % Face matrix where infinite face is ignored.
% We are ignoring the infinite face here and dealing only
% with faces where all vertices belong to the graph

for i= 1:f_dual
    zero_or_not = ismember(dual_face_info(i,:),0);
    % To check whether the face has a zero, ie, whether the infinite face
    % is involved or not.
    if zero_or_not ==0
        upd_dual_face_info = [upd_dual_face_info;dual_face_info(i,:)];
        % Now we consider only the faces of the dual graph where all the
        % vertices belong to the dual graph D
    end
end

upd_f_dual = size(upd_dual_face_info,1); % Number of faces in the updated
% dual face matrix

% Now we begin constructing the colex

m=1; %count variable
for i = 1:v_dual

    temp_vert = find(dual_adj(i,:)); % All edges connected to vertex i in D
    num_temp_vert = size(temp_vert,2);
    temp8 = dual_vert(i,:); % Coordinates of vertex i in D

    for j = 1:num_temp_vert % For every edge connected to i
        temp9 = dual_vert(temp_vert(j),:); %Coordinates of vertex j in D
        temp10 = 0.75*temp8 + 0.25*temp9; % creating a point 1/4 distance from i-j
        colex_vert(m,:) = [ temp10 i temp_vert(j) ];
        %colex vertex coords and edge where the vertex came from
        m=m+1; % Move to next vertex, m is a count on number of vertices
    end
end

for i=1:dual_num_edge % We are going to connect the two new vertices

```

```

%created at each edge i-j
sub_colex_vert = colex_vert(:,[3,4]); % Only the coordinates of colex_vert,
% last 2 columns

temp11 = ismember(sub_colex_vert,i); % returns a 1 wherever i is present
[irows,icols] = find(temp11);
%irows will give row number(which is the index of the new vertices)
%where i occurs in D

for j =1:dual_num_edge
    temp12 = ismember(sub_colex_vert,j);
    [jrows,jcols] = find(temp12);
    %jrows will give row number(which is the index of the new vertices)
    % where j occurs in D
    if i~=j
        common_row2 = intersect(irows,jrows); % Where both i and j are present.
        %common_row2 will be a 1x2 matrix, because every edge occurs
        %twice in colex_vert
        if isempty(common_row2) == 0 % If i and j are indeed connected
            v1 = common_row2(1);
            v2 = common_row2(2); % We find the row numbers where
            % i and j are present
            colex_adj(v1,v2)=1;
            colex_adj(v2,v1)=1;

            end

        end

    end

end

end

for i=1:upd_f_dual
    % We are going to connect remaining edges by iteratively proceeding in
    % each 'updated' dual face from upd_dual_face_info

    facelist1 = upd_dual_face_info(i,:); %List of vertices in a-clockwise in face i
    num_facelist1= size(facelist1,2); % Number of vertices in the face
    size_temp2 = size(colex_face_info,2); % Present number of vertices in each row

    temp23 = size_temp2 - num_facelist1; % Difference between the two

    % We are going to update the face matrix in a-clockwise direction of
    % the new colex
    if size_temp2 ~= 0 % If colex_face_info is already non empty
        if temp23 >0

```



```

        % If this face has lesser number of vertices than already existing
        % faces, make the rest of the entries in this face to be zero, so
        % that in the face matrix of the colex, all the faces have equal
        % number of vertices- zero indicating it is not a vertex
        facelist1extend = [ facelist1 zeros(1,temp23)];
        colex_face_info = [colex_face_info;facelist1extend ];

elseif temp23 <0    % If present face has more vertices
    size_temp26 = size(colex_face_info,1);

    colex_face_info = [colex_face_info zeros(size_temp26,temp23)];
else % If temp23 ==0, then we can just append current face list
%to colex_face_info
    colex_face_info = [colex_face_info;facelist1 ];

end

else % if colex_face_info is empty
    colex_face_info = [colex_face_info;facelist1 ];
end

% We are storing this in the face matrix for colex

facelist1 = [facelist1 facelist1(1)];
% adding the 1st vertex to the end to complete cycle in a-clockwise

colex_face_vertlist = [];
% For a given face i, this will store the sequence of new vertices in
% anti clockwise order that we are going to connect.

for j =1:num_facelist1 % For every vertex in the face
    %j is the current vertex, j+1 is the next vertex

    temp13 = ismember(sub_colex_vert,[facelist1(j) facelist1(j+1)],'rows');
    [jrowss,jcolss] = find(temp13);
    %jrowss will give the vertex created for j in the edge j-j+1

    temp14 = ismember(sub_colex_vert,[facelist1(j+1) facelist1(j)],'rows');
    [jrowss2,jplus1colss] = find(temp14);
    %jrowss2 will give the vertex created for j+1 in the edge j+1-j

```

```

        colex_face_vertlist = [colex_face_vertlist;jrowss;jrowss2] ;
    end

    size1 = size(colex_face_vertlist,1); % Number of vertices in this matrix

    for j=1:size1-1
        temp21 = colex_face_vertlist(j); %This is the vertex in current edge
        temp22 = colex_face_vertlist(j+1); %This is the vertex in next edge
        %in a-clockwise order

        colex_adj(temp21,temp22)=1;
        colex_adj(temp22,temp21)=1; % We are connecting these two vertices
    end

    temp23 = colex_face_vertlist(size1); % This is the last vertex
    temp24 = colex_face_vertlist(1); % This is the first vertex

    colex_adj(temp23,temp24)=1; % We are connecting the first and last vertices
    colex_adj(temp24,temp23)=1;

end

% dual_vert_all;
% dual_adj_full;
% dual_face;

```

Plotting the graphs

```

figure(2); gplot(adj_mat,vert_set,'-*') %Plots the original graph
axis([-1 7 -1 7]); %axis to be adjusted based on input and output
hold on;

figure(2); gplot(dual_adj_full,dual_vert_all, 'r-*')
axis([-1 7 -1 7]);%Plots the dual graph
figure(3); gplot(dual_adj_full,dual_vert_all, '-*')
axis([-1 7 -1 7]);%Plots the dual graph
figure(4); gplot(colex_adj,colex_vert, '-*') %Plots the colex
axis([-1 7 -1 7]);

figure(1); gplot(adj_mat,vert_set,'-*') %Plots the original graph
axis([-1 7 -1 7]);

```