

# INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY BANGALORE

PROJECT ELECTIVE  
T122-ACH AI-901

---

## Computing Homology

---

December 27, 2022

Rahul Jain (IMT2020117)



## Problem Statement

Given any input simplicial complex (up to 3-dimensional), corresponding to each 1-hole compute a representative 1-cycle and visualize all the representative 1-cycles.

## 1 Theoretical Solution

In our program we take vertices, edges and faces as input (from a .gts file). We then create a matrix corresponding to the linear transformation  $(\partial_2)$  and the linear transformation  $(\partial_1)$ . Additionally, we apply linear algebra on these two matrices to produce  $img(\partial_2)$  and  $ker(\partial_1)$ .

As we know,

$$\partial_2(\overline{v_1 v_2 v_3}) = \overline{v_1 v_2} + \overline{v_2 v_3} - \overline{v_3 v_1}$$

Now it is clear that  $C_2(K)$  is the vector space of all the 2-chains  $\implies \dim(C_2(K)) = \text{number of faces}$ . The matrix  $(\partial_2)$  is a mapping from  $C_2(K)$  to  $C_1(K)$ . We take the input faces and accordingly compute the matrix  $\partial_2$  with the faces making the columns and the edges making the rows. Then according to the equation stated above, we fill the matrix with 1, -1 and 0 accordingly.

To compute  $img(\partial_2)$  we can compute column space of  $(\partial_2)^T$  matrix and the span of column vectors thus obtained will map the image space of  $\partial_2$ . To compute  $ker(\partial_1)$  we can directly get null space of a matrix using in-built functions in the python library.

$$H_1(K) = ker(\partial_1) / Im(\partial_2)$$

Find the column space of  $img(\partial_2) \cap ker(\partial_1)$  and then find its complement in  $ker(\partial_1)$  to get the representative 1-cycles.

## 2 PseudoCode

Here is a pseudo code that better describes the approach and its steps in more detail.

---

**Algorithm 1** Computing Representative 1-Cycles

---

- 1: Create Boundary Matrix  $(\partial_1)$  and Boundary Matrix  $(\partial_2)$ .
  - 2:  $img(\partial_2) \equiv SPAN\{\partial_2.columnspace()\}$
  - 3:  $ker(\partial_1) \equiv SPAN\{\partial_1.nullspace()\}$
  - 4: Create matrix  $M \equiv (img(\partial_2) \mid ker(\partial_1))$
  - 5: Reduce matrix  $M$  to its row-reduced echelon form,  $M.rref()$
  - 6: Extract  $pivots = \{p_1, p_2, p_3 \dots\}$  from  $M.rref()$
  - 7:  $pivots \equiv \{pivots - p_i : \forall p_i \in img\partial_2\}$
  - 8: The above step will give us pivots in  $(ker\partial_1 - img\partial_2)$
  - 9: Find columns in  $ker\partial_1$  corresponding to pivots.
  - 10: Convert 1,-1,0 in pivot columns to connected path of edges.
  - 11: The pivot columns thus found are unique representative 1-cycles.
- 

## 3 Algorithm Analysis

If we analyze the above written algorithm, arithmetic operations majorly happened in computing nullspace, columnspace and row-reduced echelon form of matrix, all of these computations basically require **Gauss Elimination algorithm** to calculate row-reduced form of matrix.

Therefore, time complexity of the program can be measured by computing the number of arithmetic operations required to perform row reductions.

### 3.1 Row-echelon form

Let us recall the three conditions that we used to declare a matrix to be in reduced row-echelon form:

1. For every row, the left-most nonzero entry, if any, is a 1. We will call this the pivotal entry for the row and the column where this appears the pivotal column.
2. The pivotal column for a later (lower) row is a later (more to the right) column, if it exists. If a given row is all zeros, all later rows are also all zeros.
3. Any column containing a pivotal 1 must have all other entries in it equal to 0. This has two subparts:
  - (a) All entries in the column below the pivotal 1 are equal to 0.
  - (b) All entries in the column above the pivotal 1 are equal to 0.

### 3.2 Gauss Elimination Algorithm

```
1 h := 1 /* Initialization of the pivot row */
2 k := 1 /* Initialization of the pivot column */
3
4 while h in range(0,m) and k in range(0,n):
5     /* Find the k-th pivot: */
6     i_max := argmax (i = h ... m, abs(A[i, k]))
7     if A[i_max, k] = 0
8         /* No pivot in this column, pass to next column */
9         k := k + 1
10    else
11        swap rows(h, i_max)
12        /* Do for all rows below pivot: */
13        for i = h + 1 ... m:
14            f := A[i, k] / A[h, k]
15            /* Fill with zeros the lower part of pivot column: */
16            A[i, k] := 0
17            /* Do for all remaining elements in current row: */
18            for j = k + 1 ... n:
19                A[i, j] := A[i, j] - A[h, j] * f
20        /* Increase pivot row and column */
21        h := h + 1
22        k := k + 1
```

### 3.3 Analysis of Gauss Elimination Algorithm

Let us analyze the algorithm by computing how many steps it will take in the worst case and compute the complexity of the algorithm. Let us find out how efficient the algorithm is:

We start by defining a step, for the case of Gaussian methods a basic step is either an addition or a multiplication. We differentiate between the two because on most platforms the time needed for a multiplication is much longer than that needed for an addition. Note that we are considering floating point addition and multiplication, integer addition is generally much faster.

The number of operations required to solve a system of equations by Gaussian elimination and back substitution is the same as that required for the Gauss-Jordan method, but the Gauss-Jordan method is slightly easier to count.

We consider the cost of the elementary row operations on an  $m \times n$  matrix  $A$  augmented with  $\mathbf{b} \in R^m$ , so there are  $n + 1$  columns.

- $R_i \rightarrow cR_i$ . Each of the  $n + 1$  elements of row  $i$  must be multiplied, so cost is  $n + 1$  multiplications.
- $R_i \rightarrow R_i + cR_j$ . Each of the  $n + 1$  elements of row  $j$  must be multiplied, then each of these must be added to the corresponding element in row  $i$ . Thus the cost is  $n + 1$  multiplications and  $n + 1$  additions.
- $R_i \leftrightarrow R_j$ . With a correct implementation using pointers this operation is effectively instantaneous.

### 3.4 Time Complexity of Gaussian Elimination Algorithm

For Each row  $i$  ( $R_i$ ) from 1 to  $n$  ( $\sum_{i=1}^n$ )

If any row  $j$  below row  $i$  has non zero entries to the right of the first non zero entry in row  $i$

$R_i \leftrightarrow R_j$

$R_i \rightarrow \frac{1}{c}R_i$  where  $c$  = the first non-zero entry of row  $i$  (the pivot).

For each row  $j > i$  ( $n - i$ )

$R_j \rightarrow R_j - dR_i$  where  $d$  = the entry in row  $j$  which is directly below the pivot in row  $i$ .

If any 0 rows have appeared exchange them to the bottom of the matrix. next  $i$  [Matrix is now in REF ]

For each non zero row  $i$  ( $R_i$ ) from  $n$  to 1 ( $\sum_{i=1}^n$ )

For each  $j < i$  ( $n - i$ )

$R_j \rightarrow R_j - bR_i$  where  $b$  = the value in row  $j$  directly above the pivot in row  $i$ .

So a rough calculation counting the number of multiplications yields

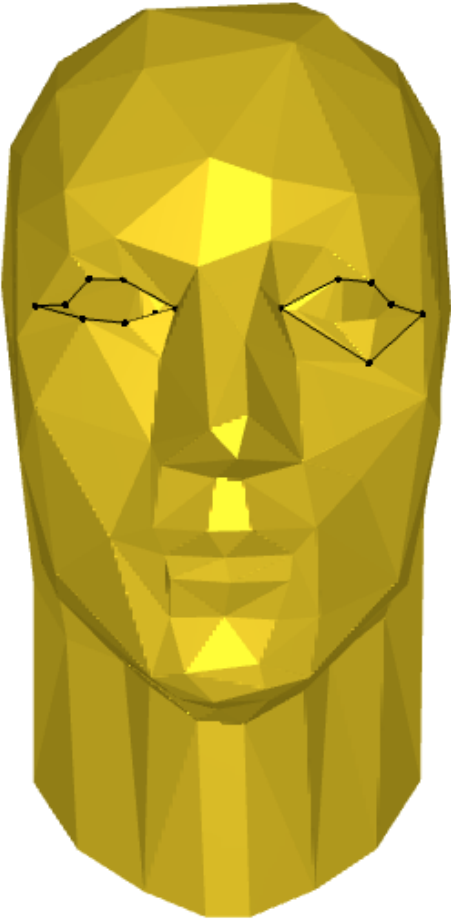
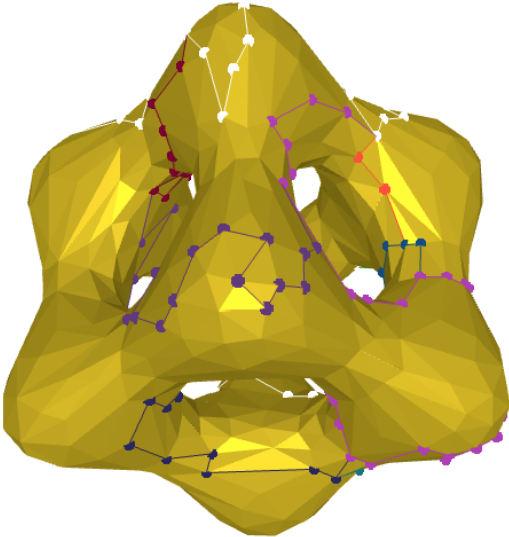
$$\begin{aligned} & \sum_{i=1}^n [(n+1) + (n-i)(n+1)] + \sum_{i=1}^n [(n-i)(n+1)] \\ &= \sum_{i=1}^n \sum_{i=1}^n (2n - 2i + 1)(n+1) \\ &= \sum_{i=1}^n 2n^2 + 3n + 1 - 2(n+1)i \\ &= 2n^3 + 3n^2 + n + n(n+1)^2 \\ &= 3n^3 + 5n^2 + 2n \end{aligned}$$

A similar rough calculation for the number of additions yields  $\sum_{i=1}^n 2(n-i)(n+1) = n^3 - n$  This is only a rough calculation. If we are considering the behaviour of the algorithm for large  $n$ , the highest term will dominate.

**We can say that the Gauss-Jordan algorithm has order  $n^3$  and write  $O(n^3)$ .**

# 4 Visualization

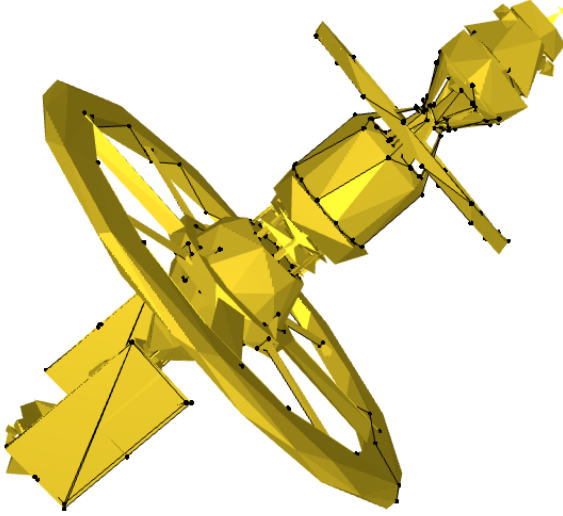
Table 1: Results for Various Simplicial Complexes

Visualization	Properties										
HEAD.GTS											
	<table><tr><td><math>\beta-1</math></td><td>2</td></tr><tr><td>Vertices</td><td>150</td></tr><tr><td>Edges</td><td>427</td></tr><tr><td>Faces</td><td>276</td></tr><tr><td>Time</td><td>377.6964 sec</td></tr></table>	$\beta-1$	2	Vertices	150	Edges	427	Faces	276	Time	377.6964 sec
$\beta-1$	2										
Vertices	150										
Edges	427										
Faces	276										
Time	377.6964 sec										
TANGLE.GTS											
	<table><tr><td><math>\beta-1</math></td><td>10</td></tr><tr><td>Vertices</td><td>593</td></tr><tr><td>Edges</td><td>1806</td></tr><tr><td>Faces</td><td>1204</td></tr><tr><td>Time</td><td>286.4399 sec</td></tr></table>	$\beta-1$	10	Vertices	593	Edges	1806	Faces	1204	Time	286.4399 sec
$\beta-1$	10										
Vertices	593										
Edges	1806										
Faces	1204										
Time	286.4399 sec										

*To be continued*

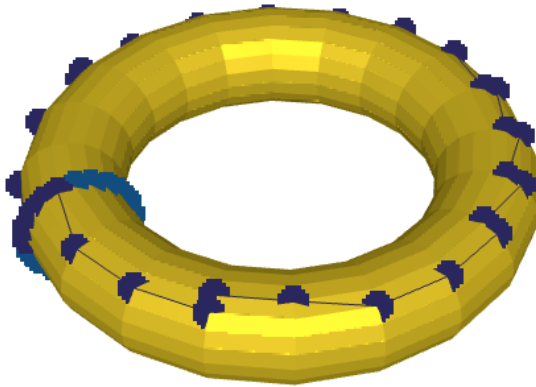
Table 1: Results for various simplicial complexes (*continued*)

SPACE STATION.GTS



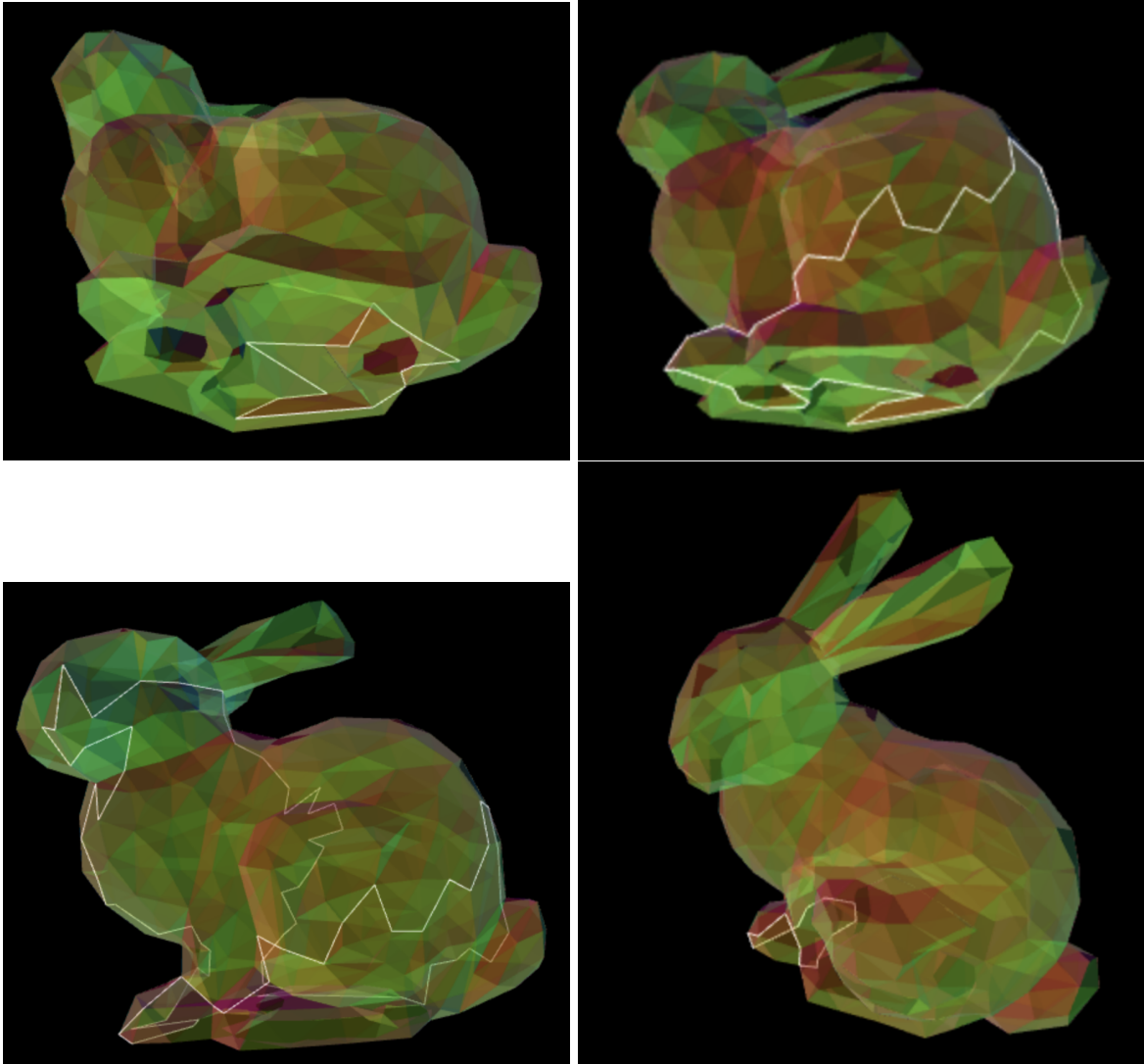
$\beta-1$	156
Vertices	1052
Edges	2476
Faces	1461
Time	201.5287 sec

TORUS.GTS

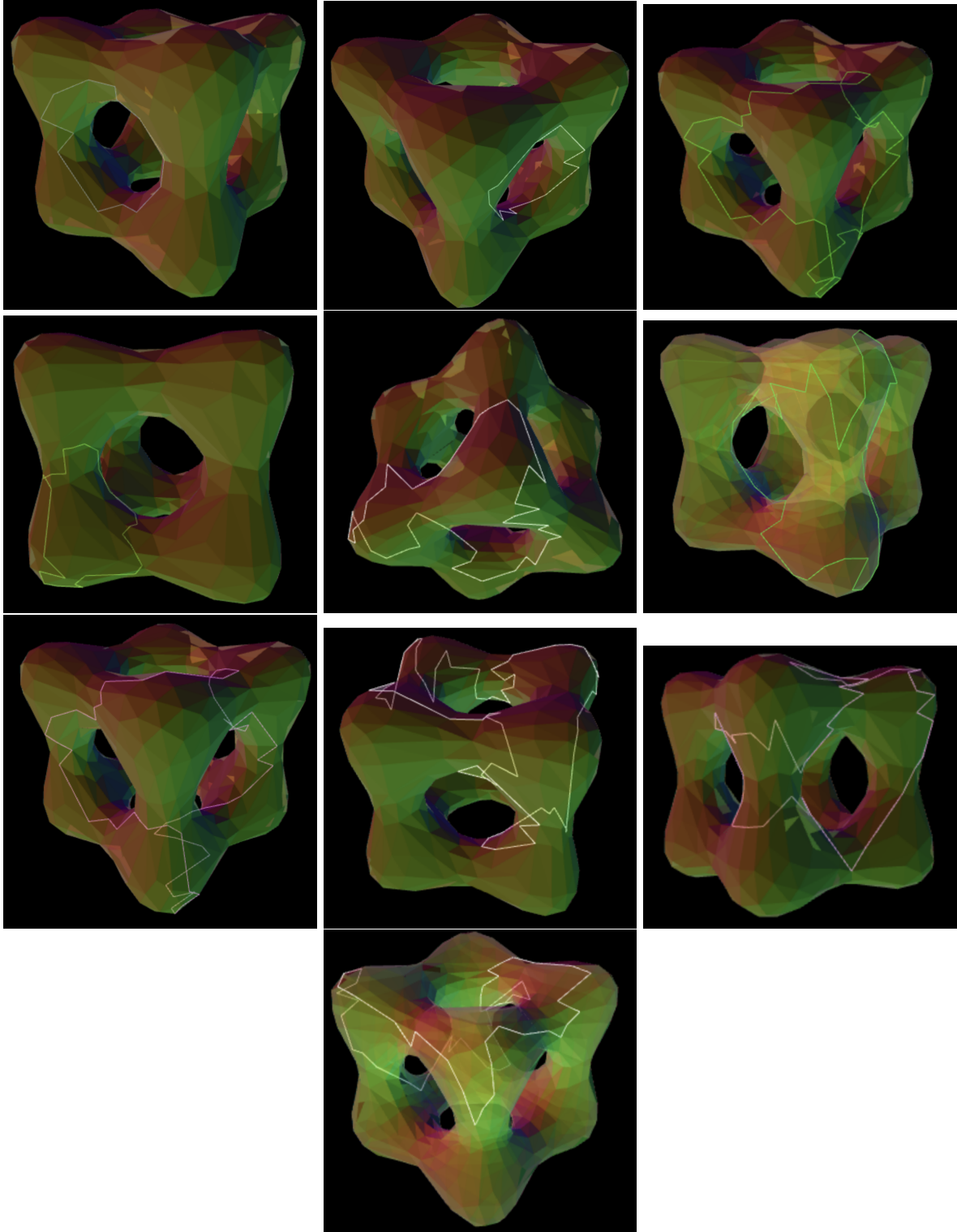


$\beta-1$	2
Vertices	400
Edges	1200
Faces	800
Time	204.8390 sec

## Visualizing 4 Representative One-Cycles of Bunny



## Visualizing 10 Representative One-Cycles of Tangle



## 5 Additional programs that assists computation of One-Cycles

### 5.1 Reducing Triangulations of Simplicial Complex

If the input GTS file has vertices/edges/faces exceeding 5000, it is better to reduce the number of triangulations via any software that deals with creation, editing and modification of mesh models.

One such software is Blender. Steps to reduce triangulations in a mesh via Blender are as follows:



1. Convert GTS file to OFF file via `off_to_gts.py`
2. OFF file can be converted using [OFF-to-OBJ](#)
3. Import the OBJ file that we got in the previous step to Blender.
4. Select Object Mode in Blender, add modifier **Triangulate** and modifier **Decimate** to reduce triangulations as per the requirement.
5. Export the object as wavefront (.obj) file while deselecting UV-Coordinates, Normals, and Materials.
6. Convert the resultant OBJ file to OFF via [OBJ-to-OFF](#)
7. Convert OFF file to GTS file via `gts_to_off.py`
8. The GTS file obtained now needs to go through `orient_simplex.cpp` to orient all the triangles of the simplicial complex.

## 5.2 Computing 0-Cycles

Before correcting the orientation of the simplicial complex, we must find out how many disconnected components exists in the mesh. By computing 0-cycles we will get to know one vertex from each disconnected-component, this way we will be able to apply orientation algorithm on the whole simplicial complex covering each and every face. Pseudocode of the algorithm is as follows:

---

### Algorithm 2 Computing Representative 0-Cycles

---

- 1: Create Boundary Matrix ( $\partial_1$ ), call it Matrix A and a diagonal Matrix of  $C_0(K)$ , call it Matrix B and join them to form Matrix A|B.
  - 2: Take RREF of the combined Matrix AB.
  - 3: Find Pivot Columns in this matrix and put them in an array.
  - 4: Now take the pivots that belongs to Matrix B from Matrix AB.
  - 5: Subtract no. of edges from those pivots so that they represent a vertex from all connected components.
  - 6: The resultant points we get are called representative 0-cycles.
- 

## 5.3 Correct the orientation of GTS file

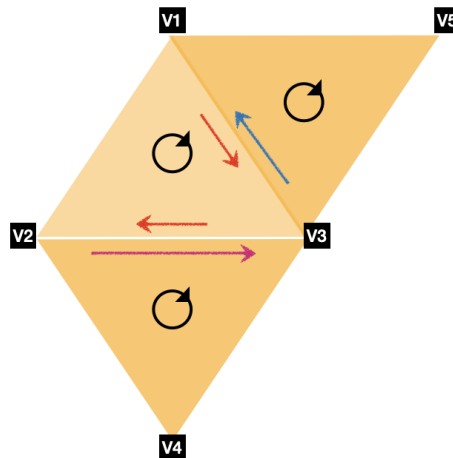


Figure 1: Correct Orientation of triangles

1. Create the following structures to store data from GTS file.

```

1 struct Vertices{
2     string coordinates;
3     int edge;
4 };
5 struct Edges{
6     int start, end; // start and end vertex
7     int faces[2] = {0}; // the two faces connected with this edge
8 };
9 struct Faces{
10     pair<int, int> edge[3] = {{0,0}, {0,0}, {0,0}}; //edge and its orientation
11     bool visited = false;
12     int chain[2] = {0}; //will define orientation of face
13 };

```

2. Every face will contain an array chain[2] which will define orientation of that face. This array will be storing two vertices of that face. Suppose for a triangle  $\overline{v_1 v_2 v_3}$ ,  
if chain =  $[v_1, v_2]$  then as per figure.1, it is anti-clockwise direction.  
else chain =  $[v_2, v_1]$  then as per figure.1, it is clockwise direction.
3. We will define orientation of any one face, which will decide the orientation of the whole simplicial complex as clockwise or anti-clockwise. holes0-list will contain the list of 0-holes, we need this so that while traversing the mesh through a modified-Depth First Search algorithm, we don't miss any disconnected component.
4. Declare a stack to keep track of visited faces during mesh traversal, and follow the algorithm below:

```

1 while (!stack.empty())
2 {
3     int top = pile.top(); // tells us the face no. we are currently at.
4     pile.pop();
5     if (faces[top].visited == true)
6         continue;
7     faces[top].visited = true;
8
9     // Use the chain to write signs of all the edges in that face.
10    for (int i = 0; i < 3; i++)
11    {
12        //If chain's start, end vertex matches with the edge, keep orientation
13        //Otherwise reverse the orientation of that edge
14    }
15
16    //Go through every edge in that face, and define opposite orientation to the ...
17    //other face connected to that edge
18    for (int i = 0; i < 3; i++)
19    {
20        //define chain[] for the other face connected to that edge.
21        //push all the connected faces to the stack.
22    }
23    //Process gets repeated untill all the faces of all disconnected components are...
24    covered.
25 }

```