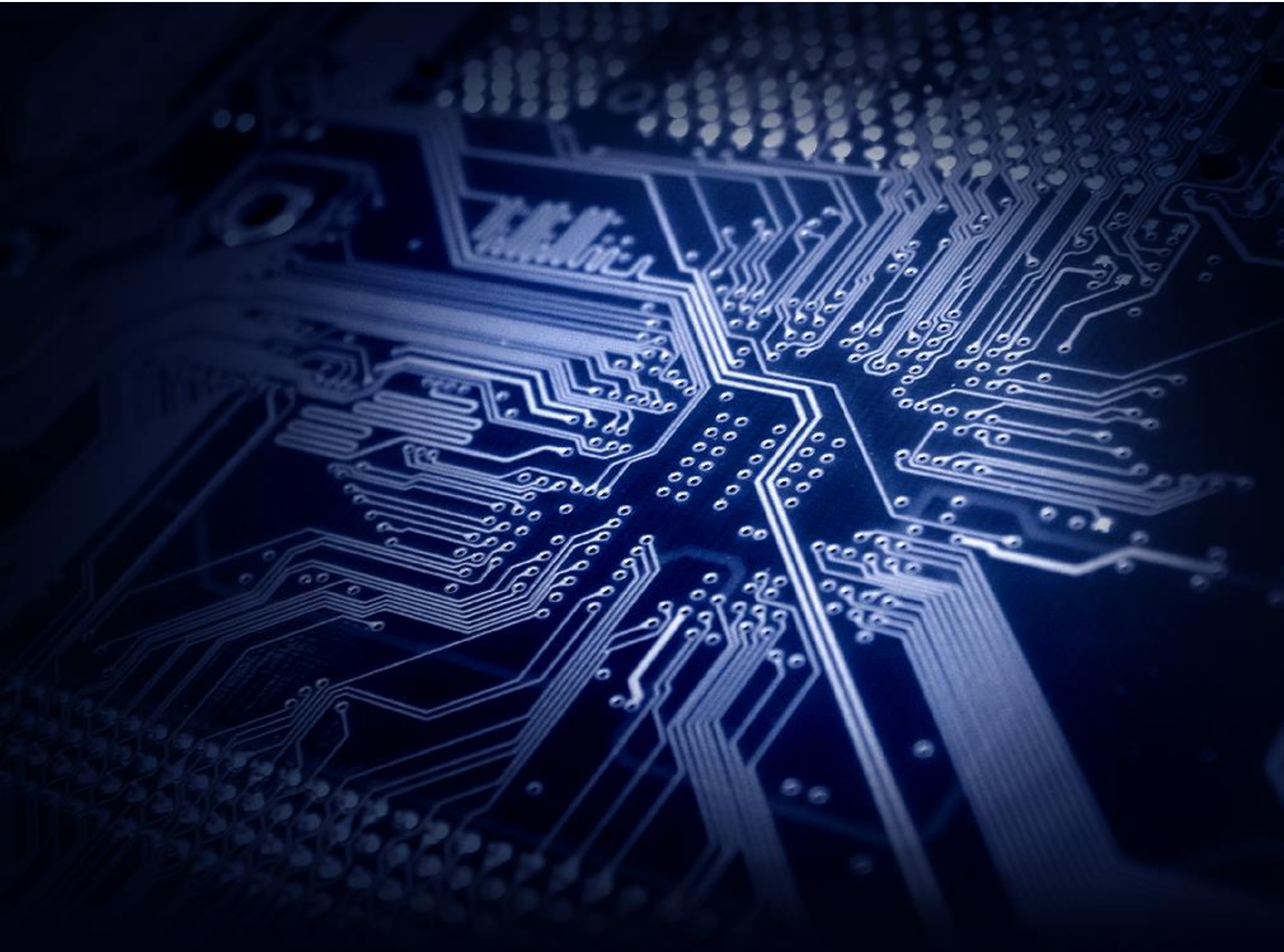


Machine Learning

Artistic Pattern Formation using Multi-Robot System

This project describes the work on multi-robot pattern formation on any kind of arbitrary target patterns with the help of optimal robot deployment, using a method that is independent of the number of robots with having utilized the least possible resources necessary to achieve the desired goal of non-intersecting pattern or design formation. Generation of uniform sets of goal positions, allocation of the multi-robots on the respective goal assignments having optimal path and fast convergence into consideration and finally smooth deployment of robots to avoid collisions using distributed collision avoidance methods to consequently provide a representative set of desired patterns. In this method, the trajectories are visually appealing in the sense of being smooth, oscillation free, and showing fast convergence having used the reliable positioning and assignment algorithms to generate both visually and convincing final pattern formation by optimizing the robots' goal positions as well as simple and smooth robot motions at the transitions of patterns.

Problem Statement

The project involves the assignment of robots to goal positions that define the final pattern, and the control of the robots to establish the formation using smooth, collision-free robot motions.

Tools Used

- *Pycharm*
- *Google Colab*
- *Python*

Important Libraries

- *Matplotlib*
- *Numpy*
- *Open CV (CV2)*

Work Distribution

NAME	MODULE	TASK ASSIGNED
M Hamza Shakoor (**Lead**)	Module – 1	Edge Detection of the shape and finding required coordinate points using Canny Algorithm. Generation of goal positions and pattern using Delaunay and Voronoi Algorithms.
M Abdullah Shahzad	Module – 2	Multi-robot assignment and convergence using Hungarian Algorithm
Saad Salman	Module – 3	Parts of Hungarian Algorithm and Collision Detection

Working of the Program

We assign a shape or figure in the form of image, select the number of available robots, then the program will automatically assign the positions to the robots to form the given pattern.
It detects main features of the given shape, then finds their coordinate points using ***Canny Edge Detection Algorithm***, forms the best optimal positions pattern using ***Delauny Triangulation and Voronoi Algorithm*** partitions the plane into the regions that are close to each other, in an artistic manner.
Hungarian Algorithm does optimal position assignment of the robots. Lastly, collision free robot motion is achieved by ***Collision Detection Algorithm.***

Methodology

The project is divided into three modules as follows:

1. *Generation of Goal Positions*

In mathematics and computational geometry, a Delaunay triangulation (also known as a Delone triangulation) for a given set P of discrete points in a general position is a triangulation DT(P) such that no point in P is inside the circumcircle of any triangle in DT(P). Delaunay triangulations maximize the minimum angle of all the angles of the triangles in the triangulation; they tend to avoid sliver triangles. The triangulation is named after Boris Delaunay for his work on this topic from 1934. For a set of points on the same line, there is no Delaunay triangulation (the notion of triangulation is degenerate for this case). For four or more points on the same circle (e.g., the vertices of a rectangle) the Delaunay triangulation is not unique: each of the two possible triangulations that split the quadrangle into two triangles satisfies the "Delaunay condition", i.e., the requirement that the circumcircles of all triangles have empty interiors. By considering circumscribed spheres, the notion of Delaunay triangulation extends to three and higher dimensions. Generalizations are possible to metrics other than Euclidean distance. However, in these cases, a Delaunay triangulation is not guaranteed to exist or be unique.

- Firstly, we will generate uniform sets of goal positions from input pattern template using algorithms like that of **Voronoi Partitioning Algorithm** or **Delaunay Triangulation Algorithm.**

- Points can be collected from the image/shape file by using two methods:

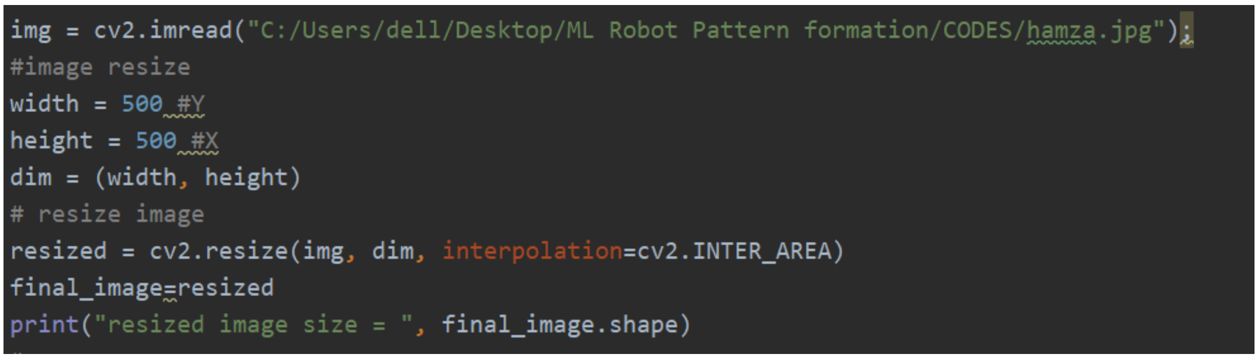
  1. For shape images or any other pattern, We used ***Canny Edge Detection Algorithm***.
  2. For Face Shapes, *dlib* library can also be used for test image that gives us facial features points.

- These methods give us final points, that are way more than our requirement. So the number of points are reduced using a special algorithm that we built. It keeps only a fraction of those points and the rest are truncated. These points are selected such that they are maximum distance apart and they form the pattern of the required shape in a best possible manner. Then these points are passed to Delaunay Triangulation and Voronoi Algorithm.

- Also, these points are used as an input to Hungarian Algorithm and then passed to Collision Detection Algorithm, that is explained later.

Canny Edge Detection

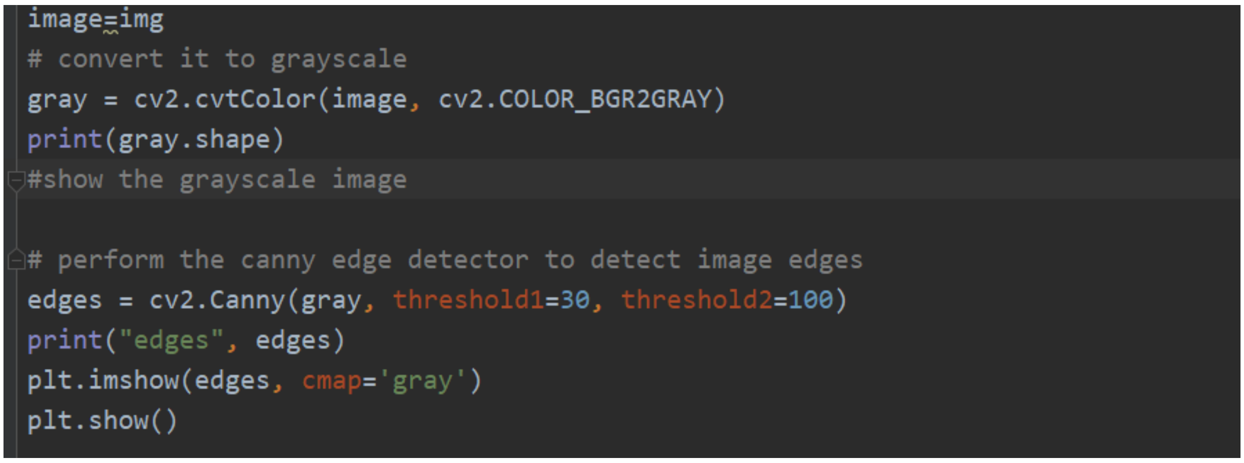
Step – 1



```
img = cv2.imread("C:/Users/dell/Desktop/ML Robot Pattern formation/CODES/hamza.jpg");
#image resize
width = 500 #Y
height = 500 #X
dim = (width, height)
# resize image
resized = cv2.resize(img, dim, interpolation=cv2.INTER_AREA)
final_image=resized
print("resized image size = ", final_image.shape)
```

➢ *Resizing of image to process it in our algorithms*

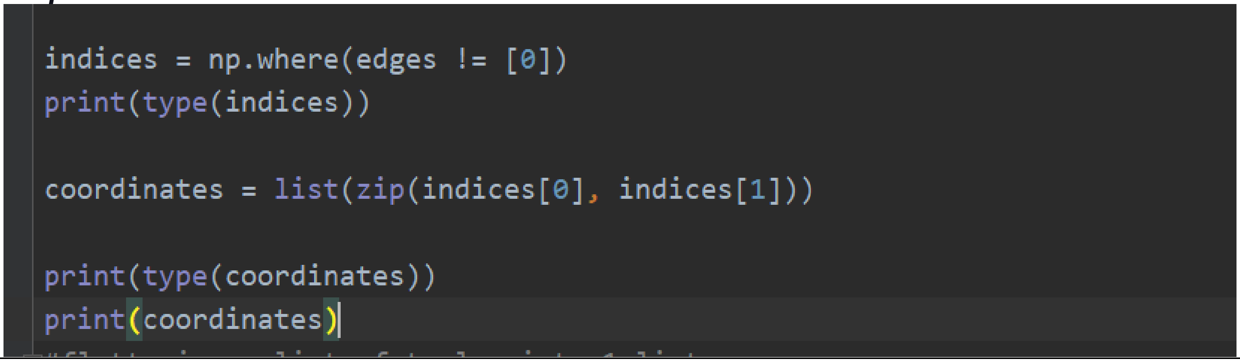
Step – 2



```
image=img
# convert it to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
print(gray.shape)
#show the grayscale image

# perform the canny edge detector to detect image edges
edges = cv2.Canny(gray, threshold1=30, threshold2=100)
print("edges", edges)
plt.imshow(edges, cmap='gray')
plt.show()
```

➢ *Generation of edge points using Canny Edge Detection Algorithm*

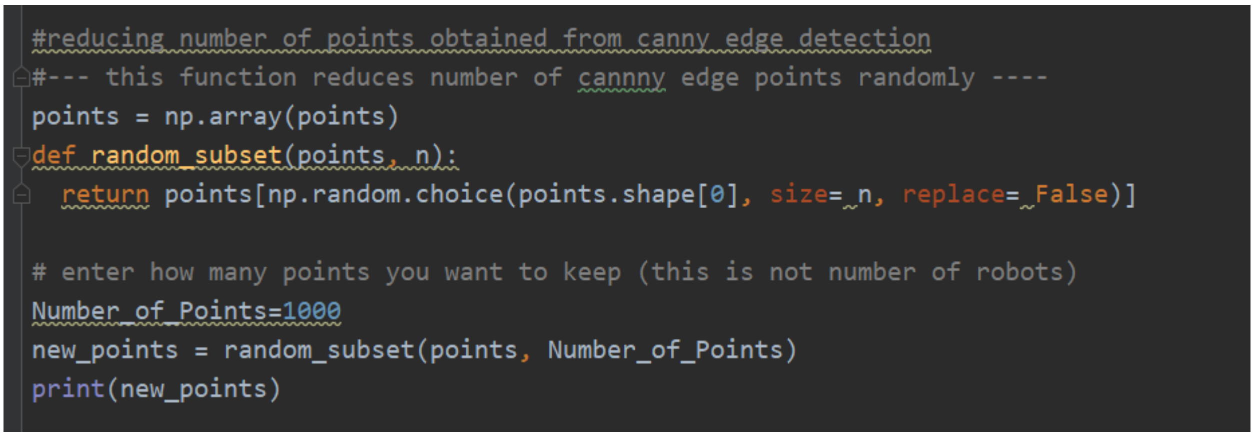
Step – 3



```
indices = np.where(edges != [0])
print(type(indices))

coordinates = list(zip(indices[0], indices[1]))

print(type(coordinates))
print(coordinates)
```

- *Converts canny edge points into coordinate points. Because every algorithm that we Used processes points in coordinate form.*

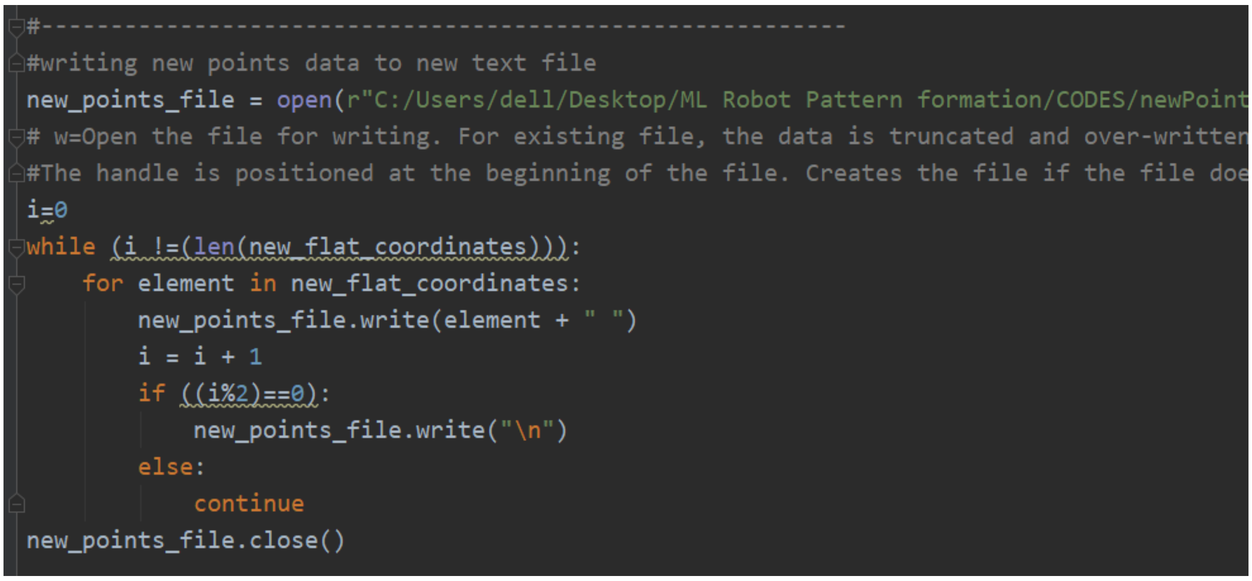
Step – 4



```
#reducing number of points obtained from canny edge detection
#--- this function reduces number of cannny edge points randomly ----
points = np.array(points)
def random_subset(points, n):
  return points[np.random.choice(points.shape[0], size= n, replace= False)]

# enter how many points you want to keep (this is not number of robots)
Number_of_Points=1000
new_points = random_subset(points, Number_of_Points)
print(new_points)
```

- *This function reduces number of obtained points*
- *Firstly, we select how many points that we want to keep from the obtained data.*
- *Because in all cases, points obtained from canny detection are in large amount of number, and we don't need that much information.*
- *So we truncate a big fraction of the points and keep only those points that form the best possible pattern and having greater distance from each other.*

Step – 5



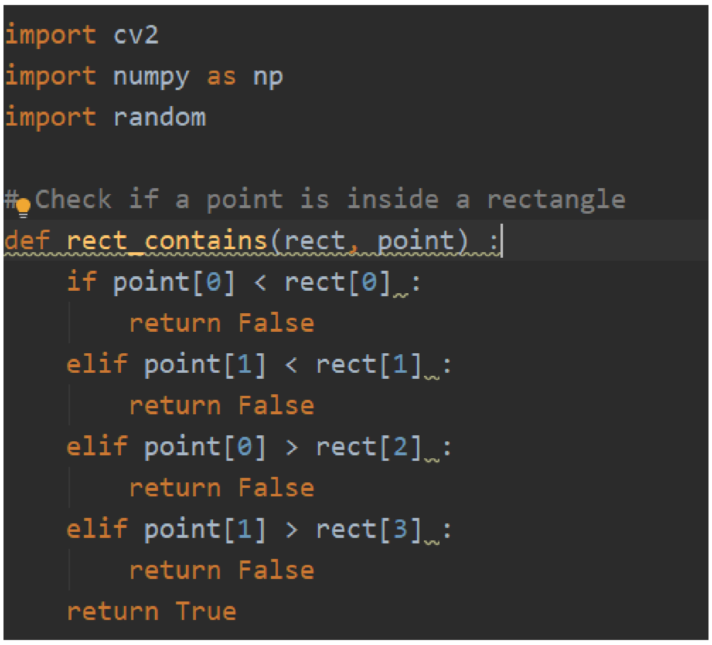
```
#-----------------------------------------------------------
#writing new points data to new text file
new_points_file = open(r"C:/Users/dell/Desktop/ML Robot Pattern formation/CODES/newPoint
# w=Open the file for writing. For existing file, the data is truncated and over-written
#The handle is positioned at the beginning of the file. Creates the file if the file doe
i=0
while (i !=(len(new_flat_coordinates))):
    for element in new_flat_coordinates:
        new_points_file.write(element + " ")
        i = i + 1
        if ((i%2)==0):
            new_points_file.write("\n")
        else:
            continue
new_points_file.close()
```

- *This part writes final obtained points into a text file.*
- *Points are in the form of coordinate points*
- *(x , y) stored in a 2D rows and coulmns form.*
- *This text file is updated everytime when a new shape is introduced to the program or the number of points is changed.*

- *It's an important file because all three modules are using it for further processing, separately.*

Delauny Triangulation and Voronoi Algorithm

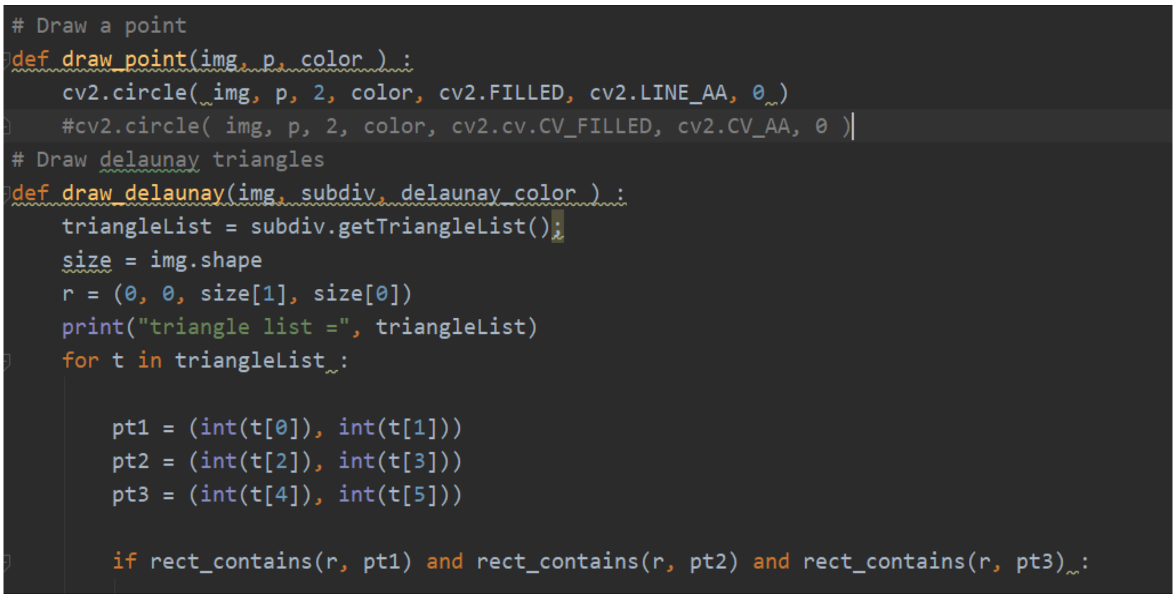
Step – 6



```
import cv2
import numpy as np
import random

#Check if a point is inside a rectangle
def rect_contains(rect, point) :
    if point[0] < rect[0] :
        return False
    elif point[1] < rect[1] :
        return False
    elif point[0] > rect[2] :
        return False
    elif point[1] > rect[3] :
        return False
    return True
```

- *Imported required libraries.*
- *Checking if the point is inside a rectangle that is passed to the delauny triangulation algorithm.*

Step – 7

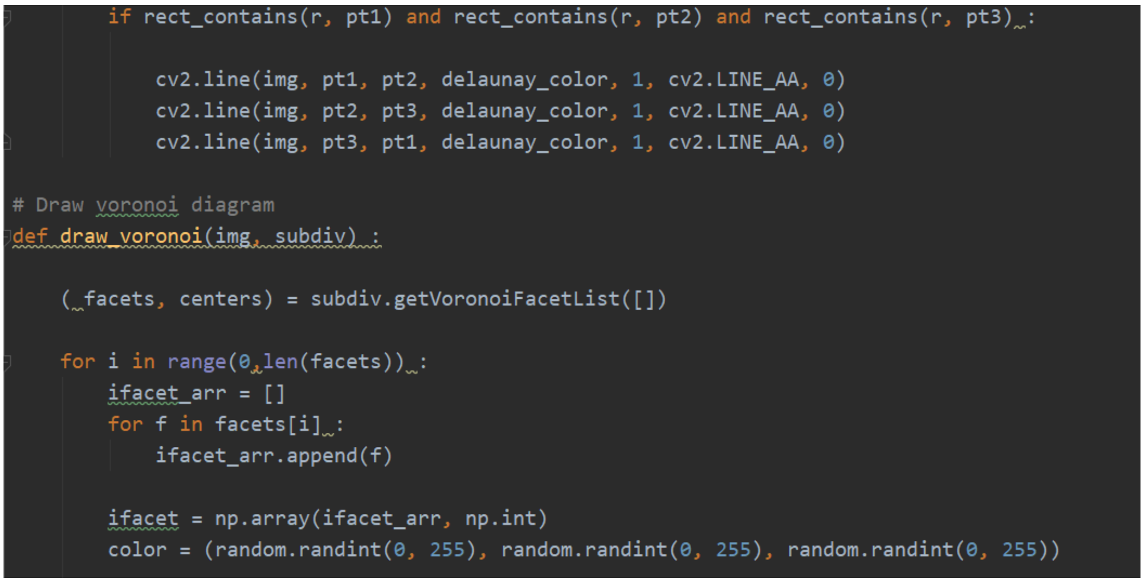


```
# Draw a point
def draw_point(img, p, color ) :
    cv2.circle( img, p, 2, color, cv2.FILLED, cv2.LINE_AA, 0 )
    #cv2.circle( img, p, 2, color, cv2.cv.CV_FILLED, cv2.CV_AA, 0 )
# Draw delaunay triangles
def draw_delaunay(img, subdiv, delaunay_color ) :
    triangleList = subdiv.getTriangleList();
    size = img.shape
    r = (0, 0, size[1], size[0])
    print("triangle list =", triangleList)
    for t in triangleList :

        pt1 = (int(t[0]), int(t[1]))
        pt2 = (int(t[2]), int(t[3]))
        pt3 = (int(t[4]), int(t[5]))

        if rect_contains(r, pt1) and rect_contains(r, pt2) and rect_contains(r, pt3) :
```

- *Defining functions, for drawing given points from text file on the given image*

- *Delauny triangulation function*
- *Where pt contains 3 vertices of one triangle (i.e x, y coordinates of vertices points)*
- *For loop, updates vertices for all triangles that need to be drawn for delauny triangulation*

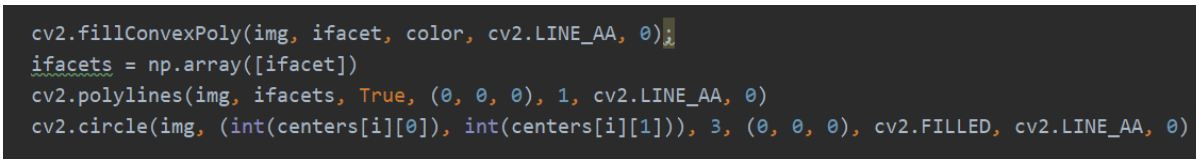
Step – 8



```
        if rect_contains(r, pt1) and rect_contains(r, pt2) and rect_contains(r, pt3) :

            cv2.line(img, pt1, pt2, delaunay_color, 1, cv2.LINE_AA, 0)
            cv2.line(img, pt2, pt3, delaunay_color, 1, cv2.LINE_AA, 0)
            cv2.line(img, pt3, pt1, delaunay_color, 1, cv2.LINE_AA, 0)

# Draw voronoi diagram
def draw_voronoi(img, subdiv) :

    ( facets, centers) = subdiv.getVoronoiFacetList([])

    for i in range(0,len(facets)) :
        ifacet_arr = []
        for f in facets[i] :
            ifacet_arr.append(f)

        ifacet = np.array(ifacet_arr, np.int)
        color = (random.randint(0, 255), random.randint(0, 255), random.randint(0, 255))
```

- *In the same function of delauny, now we are connecting the points of vertices of all the traingles to form a pattern*
- *Then there is voronoi diagram function.*

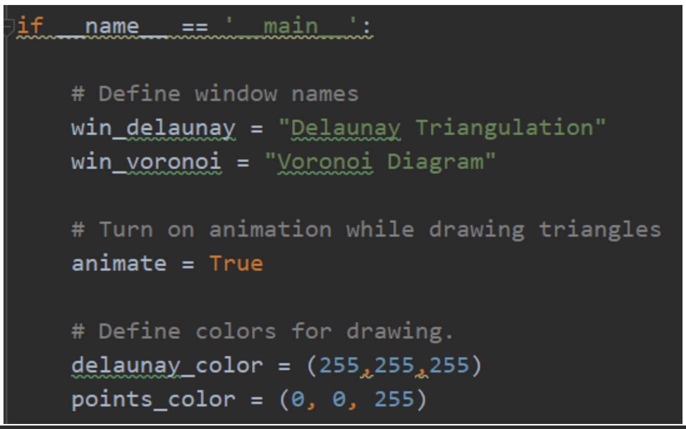
Step – 9



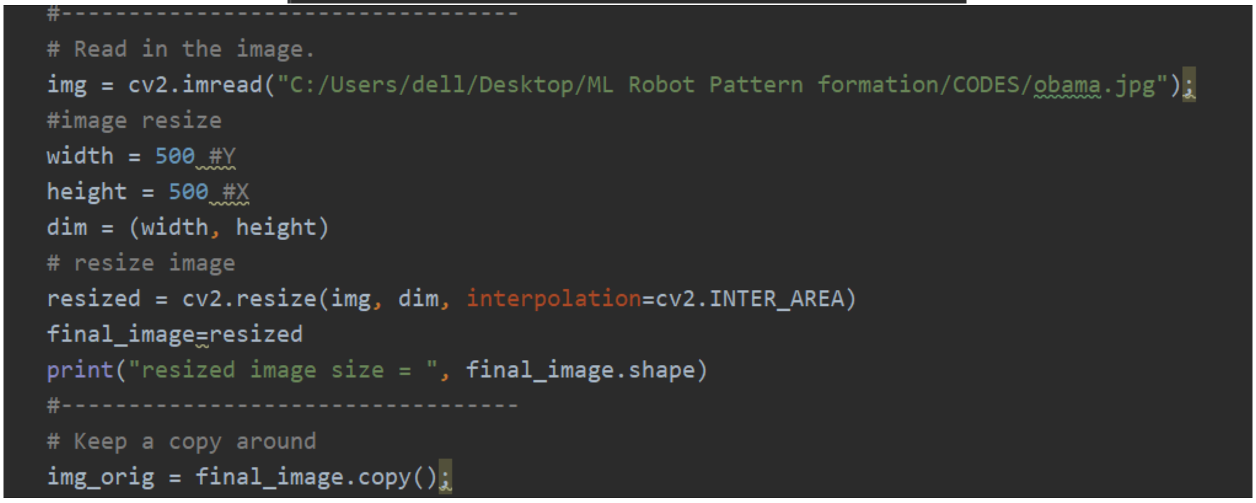
```
cv2.fillConvexPoly(img, ifacet, color, cv2.LINE_AA, 0);
ifacets = np.array([ifacet])
cv2.polylines(img, ifacets, True, (0, 0, 0), 1, cv2.LINE_AA, 0)
cv2.circle(img, (int(centers[i][0]), int(centers[i][1])), 3, (0, 0, 0), cv2.FILLED, cv2.LINE_AA, 0)
```

- *This function draws voronoi diagram by using traingles obtained from delauny.*
- *Each triangle is represented by a random different color.*

Step – 10



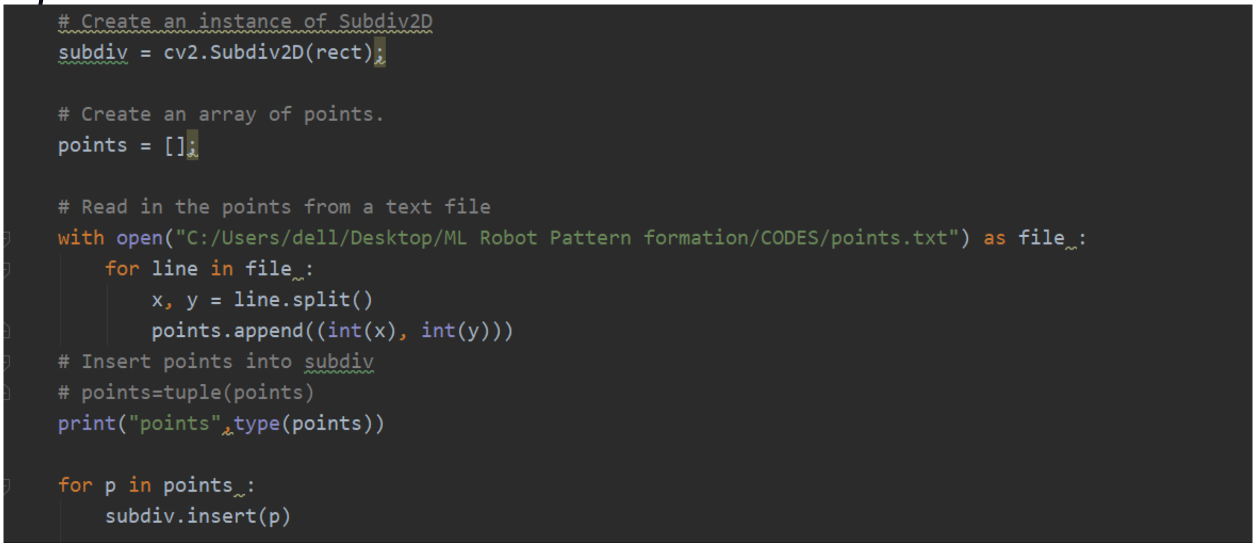
```
if __name__ == '__main__':

    # Define window names
    win_delaunay = "Delaunay Triangulation"
    win_voronoi = "Voronoi Diagram"

    # Turn on animation while drawing triangles
    animate = True

    # Define colors for drawing.
    delaunay_color = (255,255,255)
    points_color = (0, 0, 255)
```



```
#---------------------------------
# Read in the image.
img = cv2.imread("C:/Users/dell/Desktop/ML Robot Pattern formation/CODES/obama.jpg");
#image resize
width = 500 #Y
height = 500 #X
dim = (width, height)
# resize image
resized = cv2.resize(img, dim, interpolation=cv2.INTER_AREA)
final_image=resized
print("resized image size = ", final_image.shape)
#---------------------------------
# Keep a copy around
img_orig = final_image.copy();
```

- *Now we are importing the image*
- *Then resizing it (it reshaping is done according to the points that are provided by previous functions for better visual experience.*

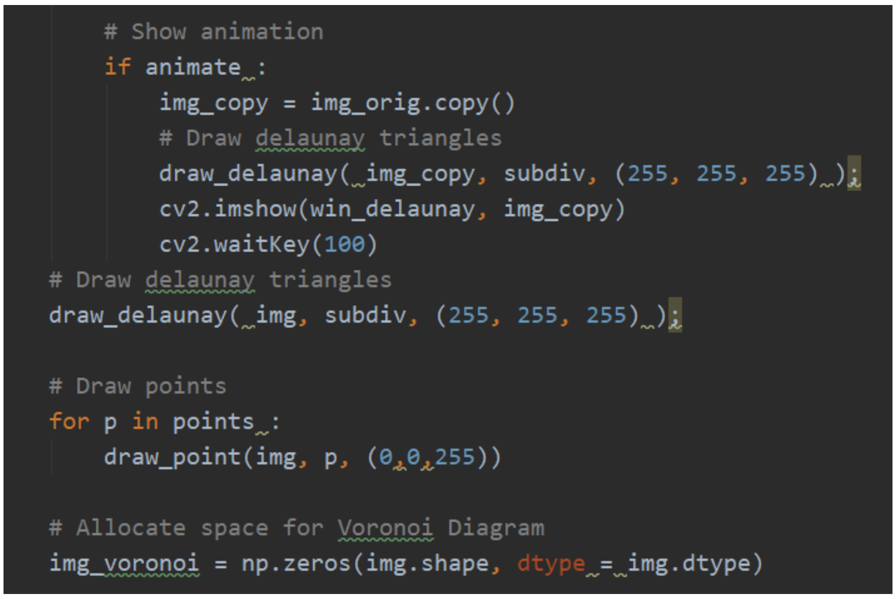
Step – 11



```
# Create an instance of Subdiv2D
subdiv = cv2.Subdiv2D(rect);

# Create an array of points.
points = [];

# Read in the points from a text file
with open("C:/Users/dell/Desktop/ML Robot Pattern formation/CODES/points.txt") as file :
    for line in file :
        x, y = line.split()
        points.append((int(x), int(y)))
# Insert points into subdiv
# points=tuple(points)
print("points",type(points))

for p in points :
    subdiv.insert(p)
```

- *Creating s subdiv2D object from given final points and passing them to delauny function*
- *Forms triangle from vertices*

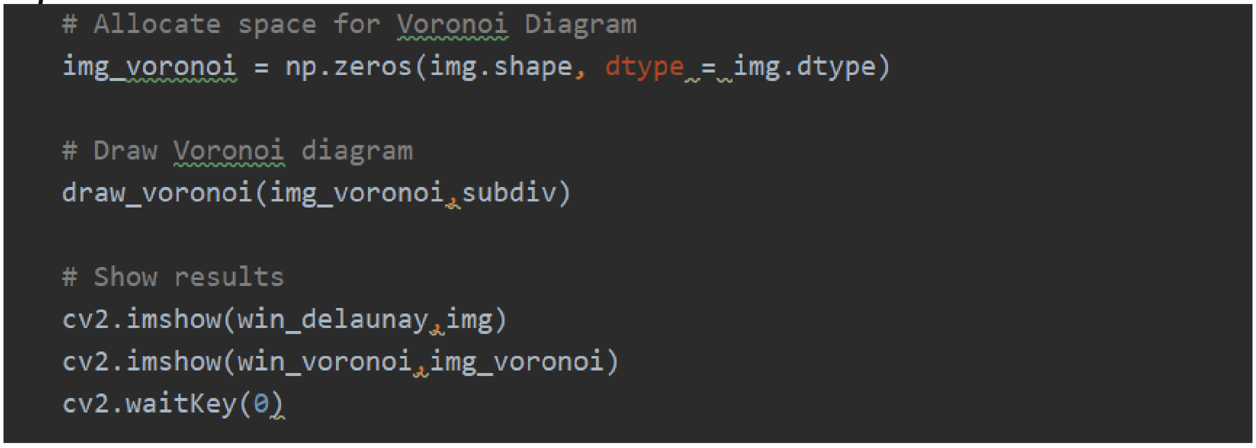
Step – 12



```
    # Show animation
    if animate :
        img_copy = img_orig.copy()
        # Draw delaunay triangles
        draw_delaunay( img_copy, subdiv, (255, 255, 255) );
        cv2.imshow(win_delaunay, img_copy)
        cv2.waitKey(100)
# Draw delaunay triangles
draw_delaunay( img, subdiv, (255, 255, 255) );

# Draw points
for p in points :
    draw_point(img, p, (0,0,255))

# Allocate space for Voronoi Diagram
img_voronoi = np.zeros(img.shape, dtype = img.dtype)
```

- ➢ *Drawing delauny triangulation by calling the previously defined function.*
- ➢ *And animating the process*
- ➢ *Then drawing points as well, on the triangulation plot*

Step – 13



```
# Allocate space for Voronoi Diagram
img_voronoi = np.zeros(img.shape, dtype = img.dtype)

# Draw Voronoi diagram
draw_voronoi(img_voronoi,subdiv)

# Show results
cv2.imshow(win_delaunay,img)
cv2.imshow(win_voronoi,img_voronoi)
cv2.waitKey(0)
```

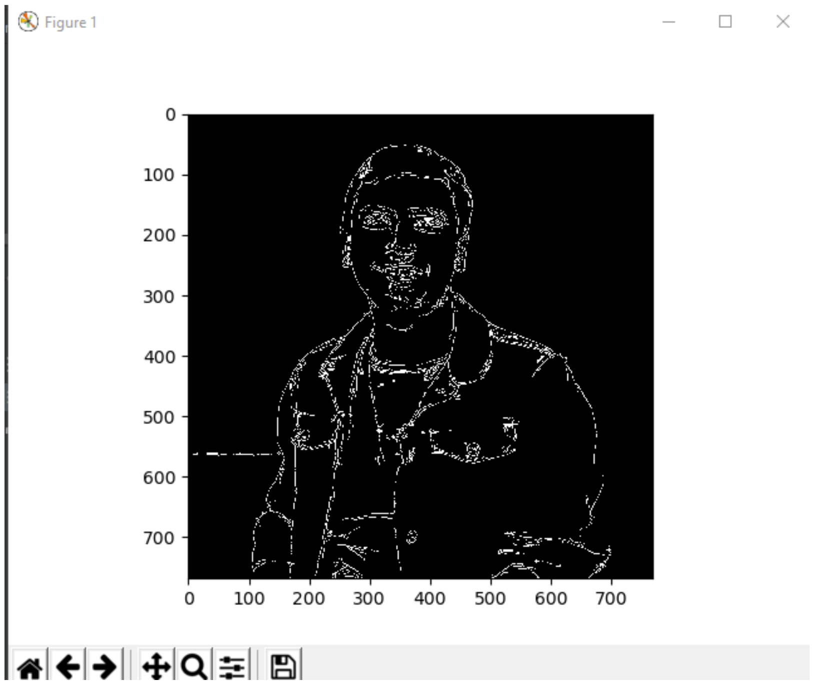
- ➢ *Allocating space for voronoi diagram*
- ➢ *Calling voronoi function*
- ➢ *Then plotting delauny plot and voroni diagram using cv2 (open CV) functions*

Output and Results

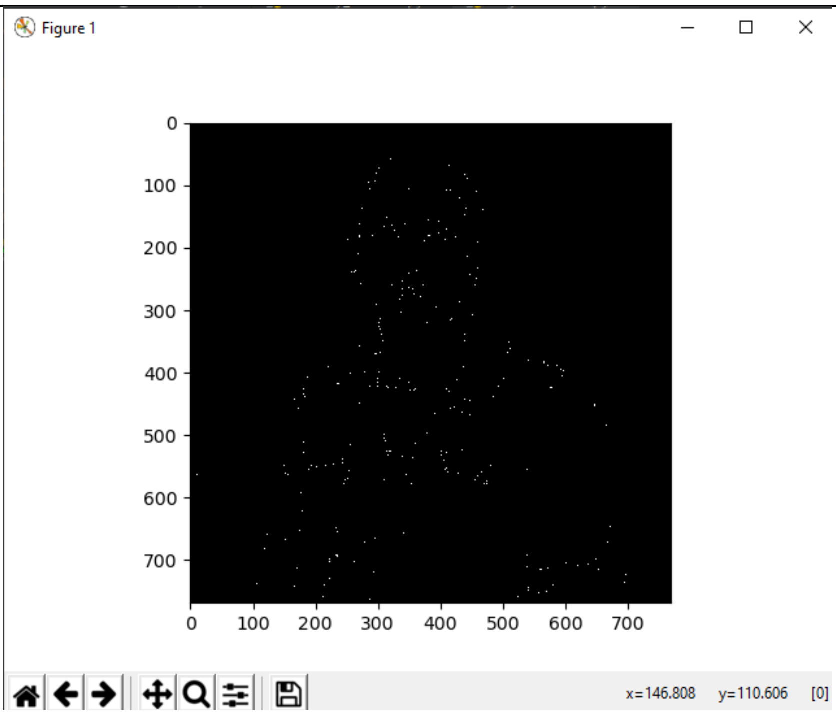
Canny Edge Detection

Test 1: Face/Body Shape

Given Image

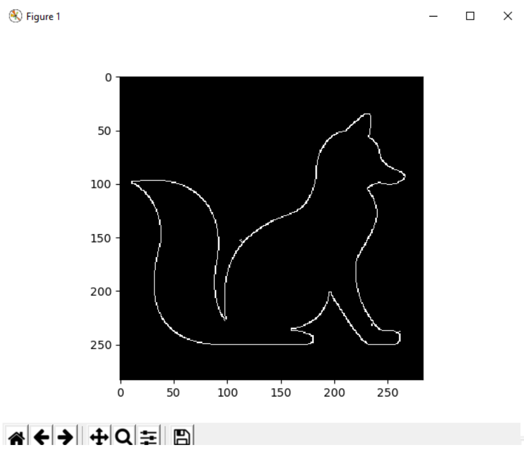


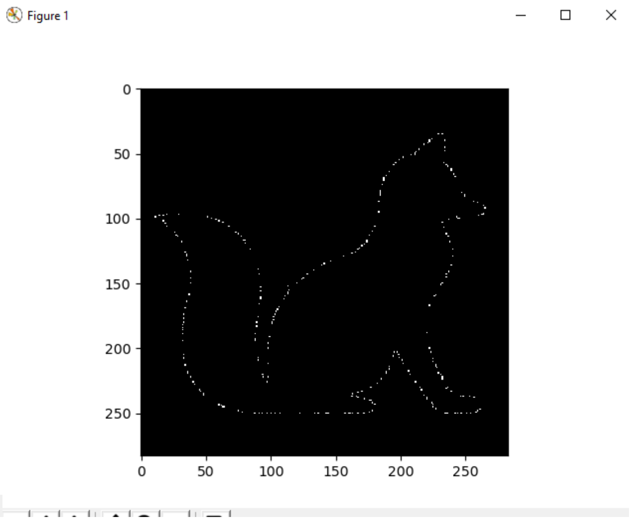
Full points edge detection



Selective points detection after truncation

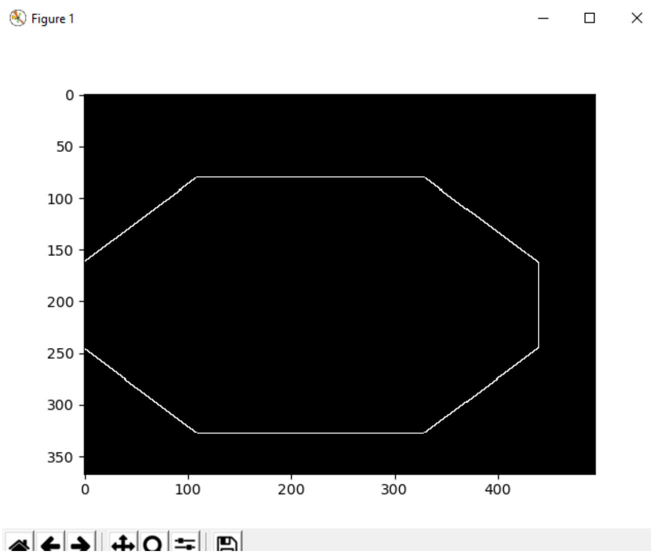
Test 2: Animal Shape

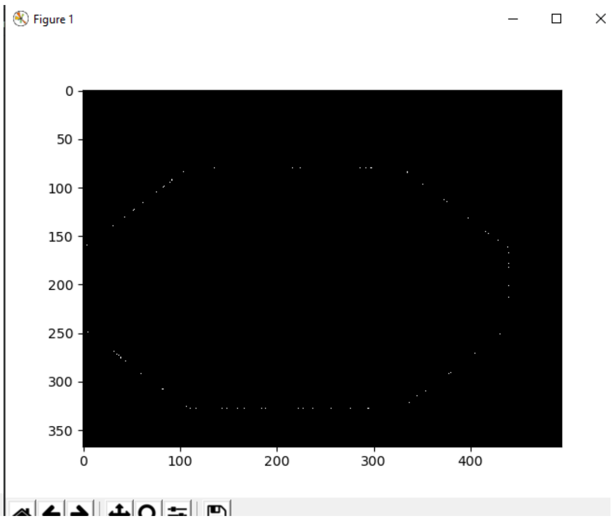




Full points edge detection ***Selective points detection***

Test 3: Mathematical Shape:

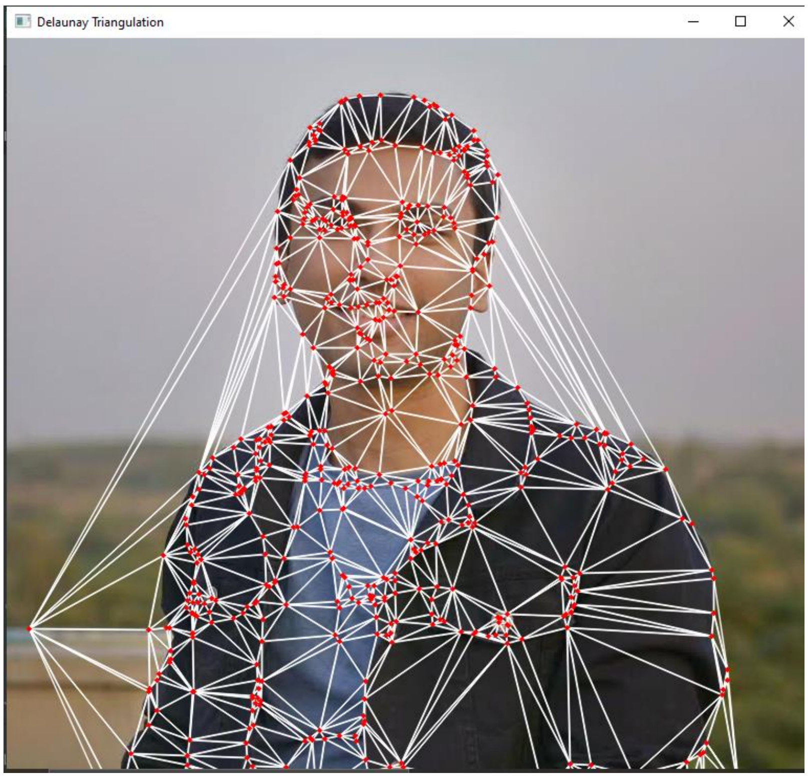




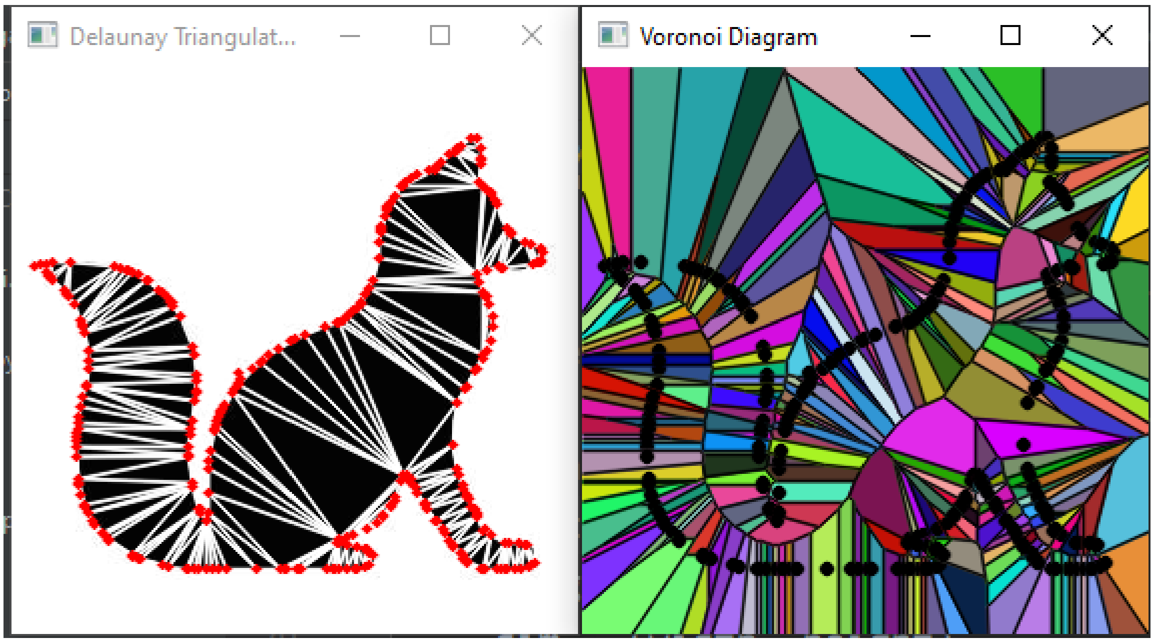
Full points edge detection ***Selective points detection***

Delaunay Triangulation And Voronoi Diagrams

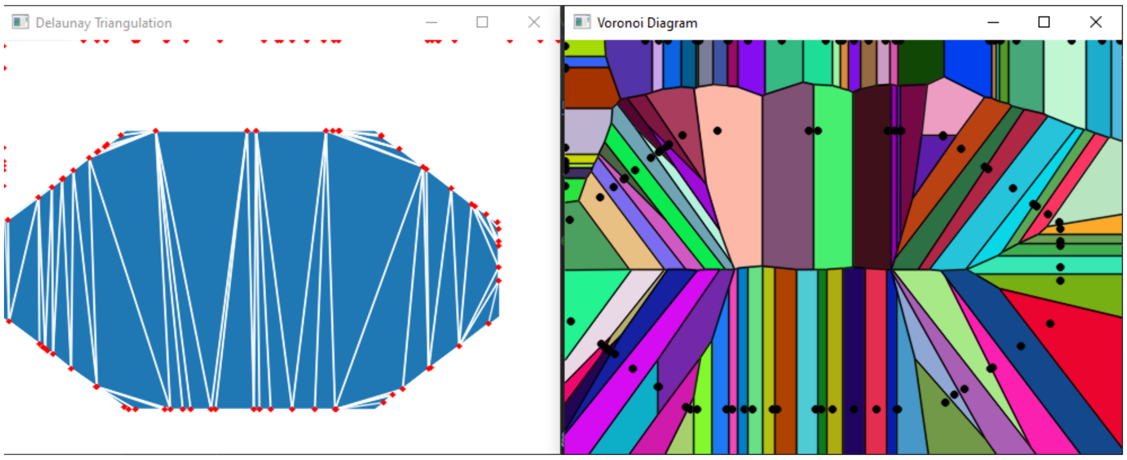
Test 1: Human Shape



Test 2: Fox

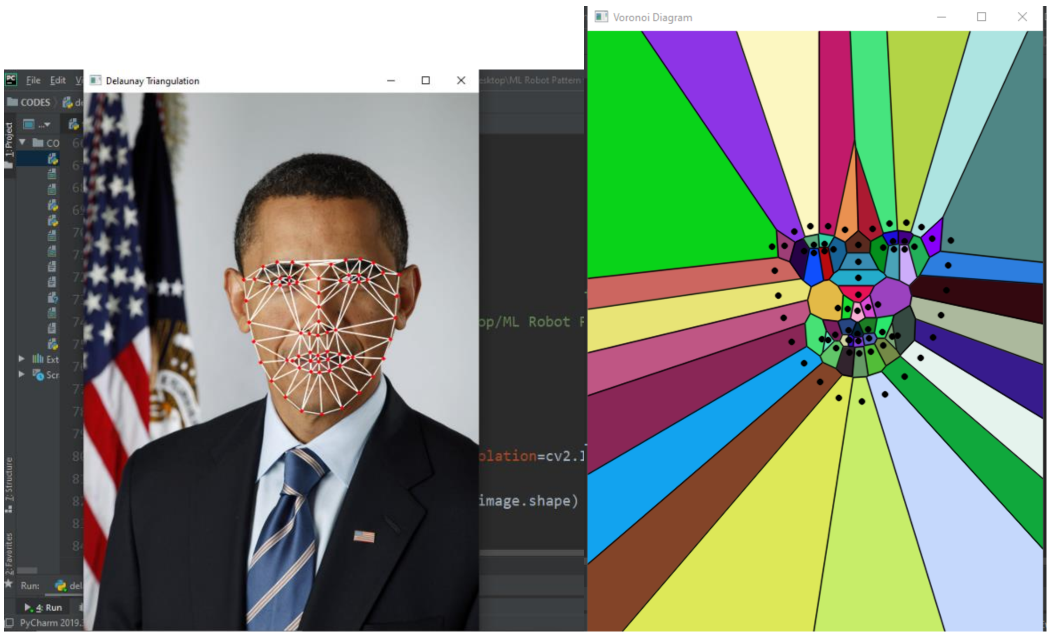


Test 3: Octagon



Test 4: Obama

(this particular example points were picked using dlib, for face feature extraction only)



2. ***Assignment of Optimal Locations for Robots***

Introduction - Although an assignment problem can be formulated as a linear programming problem, it is solved by a special method known as Hungarian Method because of its special structure. If the time of completion or the costs corresponding to every assignment is written down in a matrix form, it is referred to as a Cost matrix. The Hungarian Method is based on the principle that if a constant is added to every element of a row and/or a column of cost matrix, the optimum solution of the resulting assignment problem is the same as the original problem and vice versa. The original cost matrix can be reduced to another cost matrix by adding constants to the elements of rows and columns where the total cost or the total completion time of an assignment is zero. Since the optimum solution remains unchanged after this reduction, this assignment is also the optimum solution of the original problem. If the objective is to maximize the effectiveness through Assignment, Hungarian Method can be applied to a revised cost matrix obtained from the original matrix.

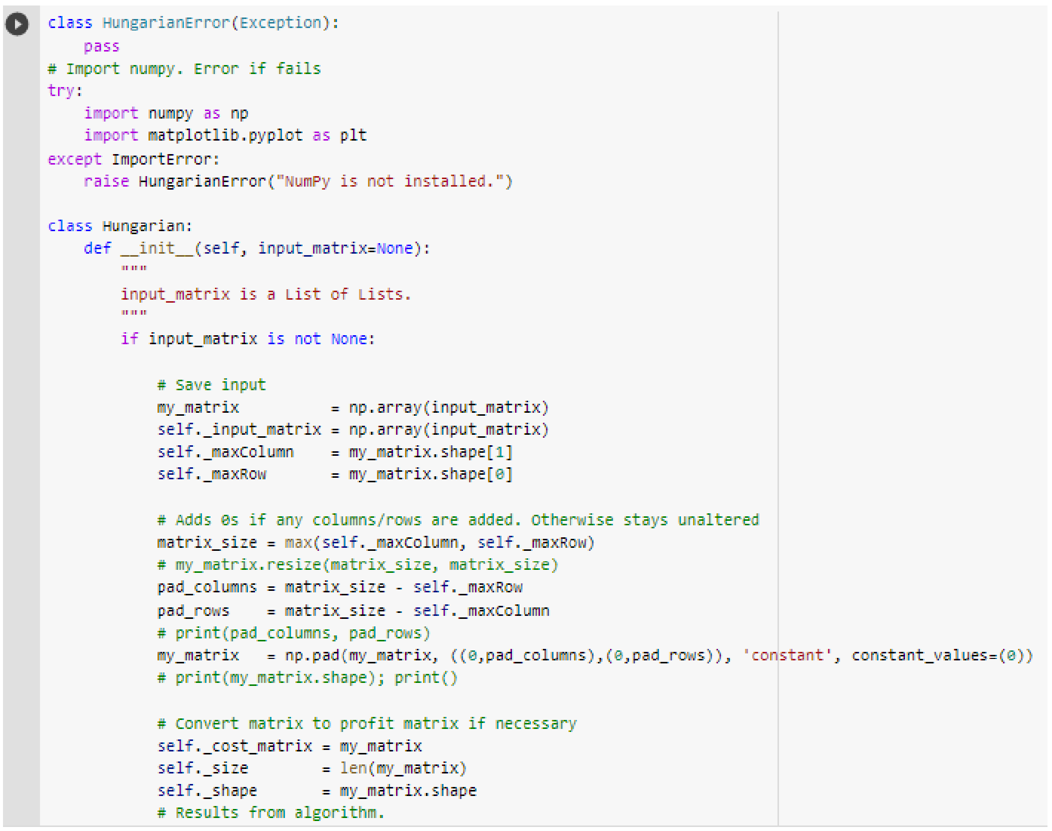
- We will be receiving the boundary points of the required pattern with the help of Voronoi Partitioning scheme and Delaunay Triangulation Algorithm.

- The received points will then be added and arranged in the form of Matrix to be optimized for the optimal assignment.

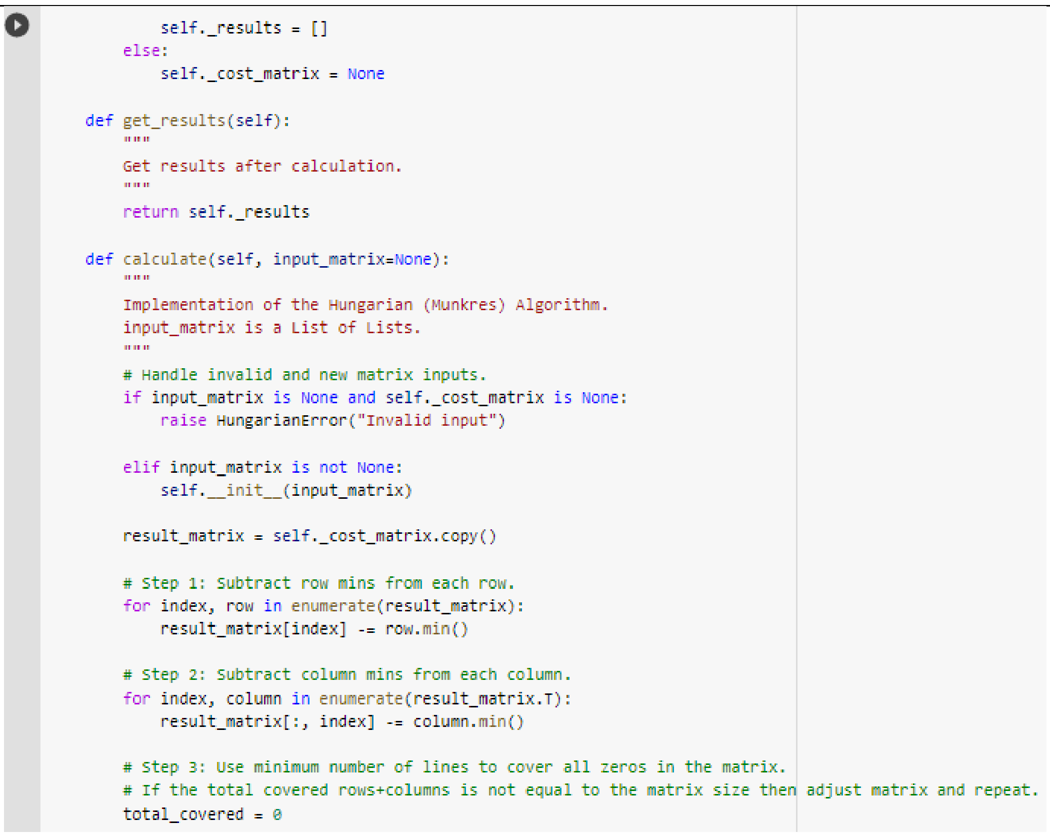
- From the consecutive row and column reduction, each robot/operator gets it assigned best value to get to.
- Then according to our resources of the robots, the assignment is made so that we achieve the pattern required or at least get close most results.
- Robots will then be driven towards the goal positions in an iterative process, by performing a multi-robot goal assignment based on the **Hungarian Algorithm** which leads to short paths and fast convergence of robots to the goal.

Hungarian Algorithm

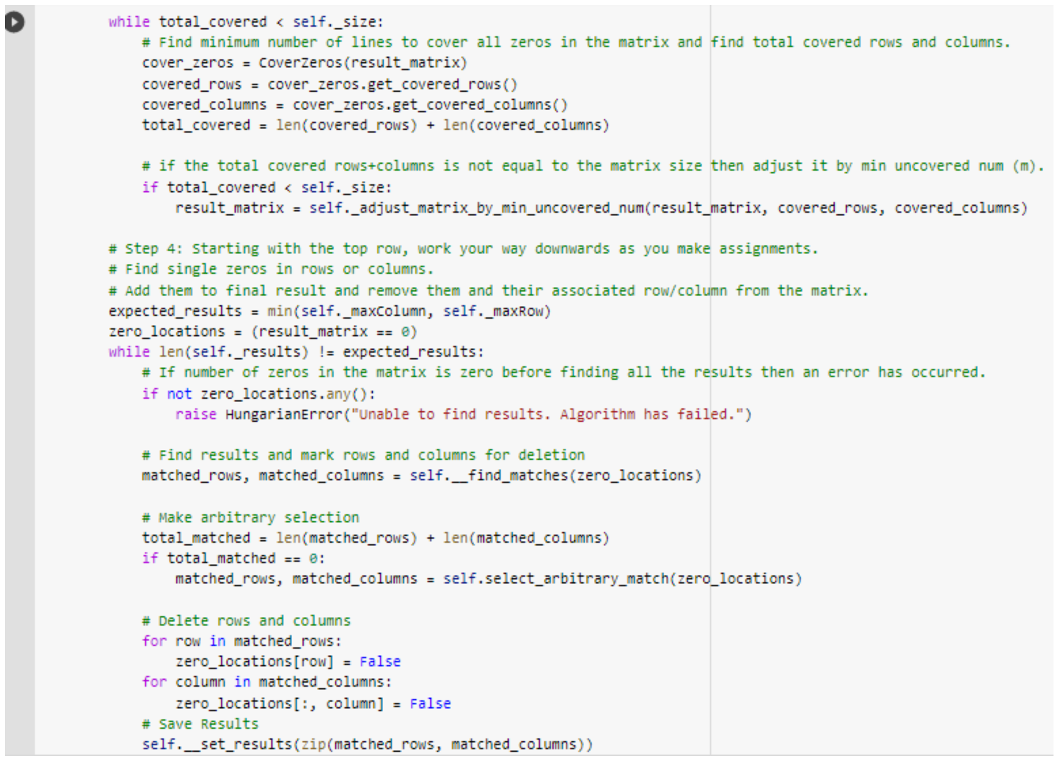
Step – 1



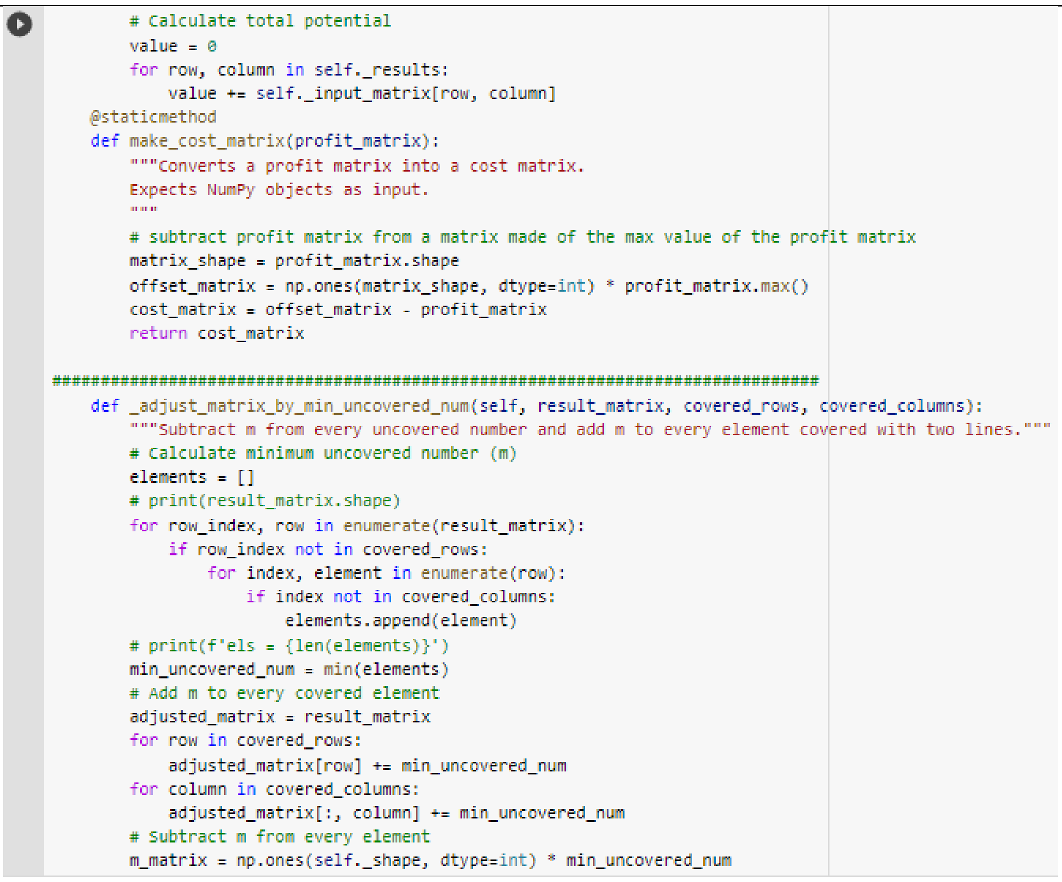
```
class HungarianError(Exception):
    pass
# Import numpy. Error if fails
try:
    import numpy as np
    import matplotlib.pyplot as plt
except ImportError:
    raise HungarianError("NumPy is not installed.")

class Hungarian:
    def __init__(self, input_matrix=None):
        """
        input_matrix is a List of Lists.
        """
        if input_matrix is not None:

            # Save input
            my_matrix          = np.array(input_matrix)
            self._input_matrix = np.array(input_matrix)
            self._maxColumn    = my_matrix.shape[1]
            self._maxRow       = my_matrix.shape[0]

            # Adds 0s if any columns/rows are added. Otherwise stays unaltered
            matrix_size = max(self._maxColumn, self._maxRow)
            # my_matrix.resize(matrix_size, matrix_size)
            pad_columns = matrix_size - self._maxRow
            pad_rows    = matrix_size - self._maxColumn
            # print(pad_columns, pad_rows)
            my_matrix   = np.pad(my_matrix, ((0,pad_columns),(0,pad_rows)), 'constant', constant_values=(0))
            # print(my_matrix.shape); print()

            # Convert matrix to profit matrix if necessary
            self._cost_matrix = my_matrix
            self._size        = len(my_matrix)
            self._shape       = my_matrix.shape
            # Results from algorithm.
```



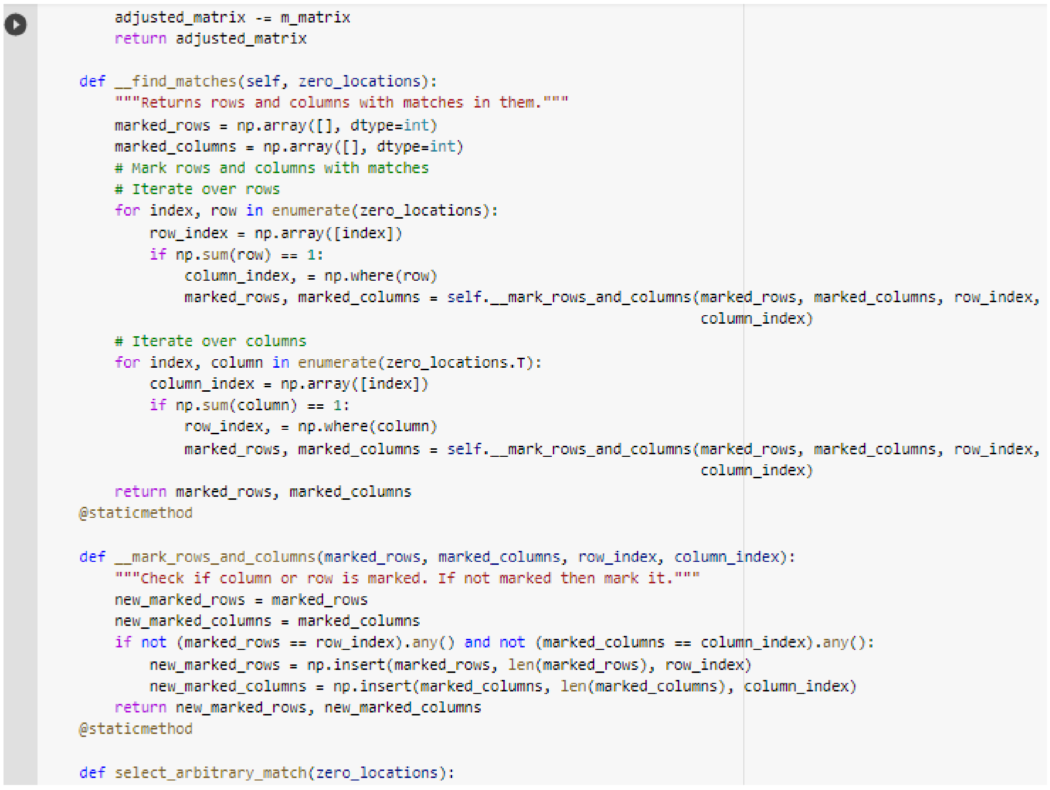
```python
            self._results = []
        else:
            self._cost_matrix = None

    def get_results(self):
        """
        Get results after calculation.
        """
        return self._results

    def calculate(self, input_matrix=None):
        """
        Implementation of the Hungarian (Munkres) Algorithm.
        input_matrix is a List of Lists.
        """
        # Handle invalid and new matrix inputs.
        if input_matrix is None and self._cost_matrix is None:
            raise HungarianError("Invalid input")

        elif input_matrix is not None:
            self.__init__(input_matrix)

        result_matrix = self._cost_matrix.copy()

        # Step 1: Subtract row mins from each row.
        for index, row in enumerate(result_matrix):
            result_matrix[index] -= row.min()

        # Step 2: Subtract column mins from each column.
        for index, column in enumerate(result_matrix.T):
            result_matrix[:, index] -= column.min()

        # Step 3: Use minimum number of lines to cover all zeros in the matrix.
        # If the total covered rows+columns is not equal to the matrix size then adjust matrix and repeat.
        total_covered = 0
```



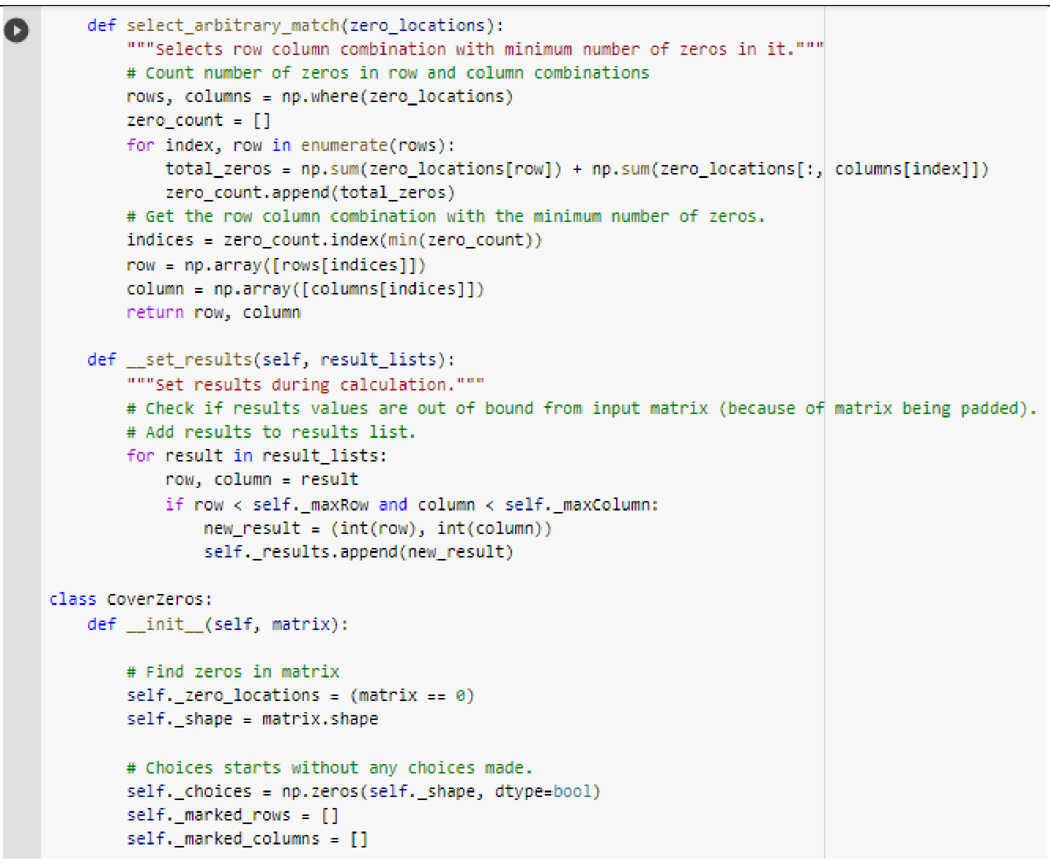
```python
        while total_covered < self._size:
            # Find minimum number of lines to cover all zeros in the matrix and find total covered rows and columns.
            cover_zeros = CoverZeros(result_matrix)
            covered_rows = cover_zeros.get_covered_rows()
            covered_columns = cover_zeros.get_covered_columns()
            total_covered = len(covered_rows) + len(covered_columns)

            # if the total covered rows+columns is not equal to the matrix size then adjust it by min uncovered num (m).
            if total_covered < self._size:
                result_matrix = self._adjust_matrix_by_min_uncovered_num(result_matrix, covered_rows, covered_columns)

        # Step 4: Starting with the top row, work your way downwards as you make assignments.
        # Find single zeros in rows or columns.
        # Add them to final result and remove them and their associated row/column from the matrix.
        expected_results = min(self._maxColumn, self._maxRow)
        zero_locations = (result_matrix == 0)
        while len(self._results) != expected_results:
            # If number of zeros in the matrix is zero before finding all the results then an error has occurred.
            if not zero_locations.any():
                raise HungarianError("Unable to find results. Algorithm has failed.")

            # Find results and mark rows and columns for deletion
            matched_rows, matched_columns = self.__find_matches(zero_locations)

            # Make arbitrary selection
            total_matched = len(matched_rows) + len(matched_columns)
            if total_matched == 0:
                matched_rows, matched_columns = self.select_arbitrary_match(zero_locations)

            # Delete rows and columns
            for row in matched_rows:
                zero_locations[row] = False
            for column in matched_columns:
                zero_locations[:, column] = False
            # Save Results
            self.__set_results(zip(matched_rows, matched_columns))
```



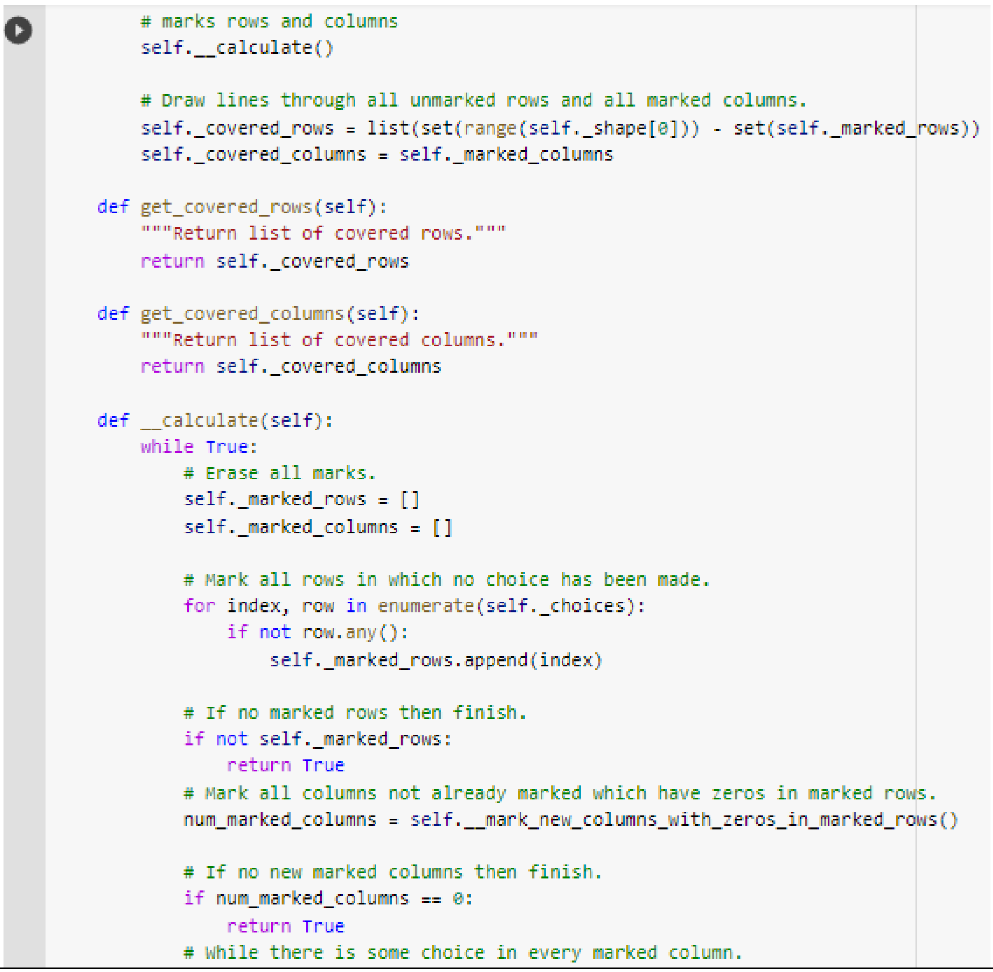
```
        # Calculate total potential
        value = 0
        for row, column in self._results:
            value += self._input_matrix[row, column]
    @staticmethod
    def make_cost_matrix(profit_matrix):
        """Converts a profit matrix into a cost matrix.
        Expects NumPy objects as input.
        """
        # subtract profit matrix from a matrix made of the max value of the profit matrix
        matrix_shape = profit_matrix.shape
        offset_matrix = np.ones(matrix_shape, dtype=int) * profit_matrix.max()
        cost_matrix = offset_matrix - profit_matrix
        return cost_matrix

################################################################################
    def _adjust_matrix_by_min_uncovered_num(self, result_matrix, covered_rows, covered_columns):
        """Subtract m from every uncovered number and add m to every element covered with two lines."""
        # Calculate minimum uncovered number (m)
        elements = []
        # print(result_matrix.shape)
        for row_index, row in enumerate(result_matrix):
            if row_index not in covered_rows:
                for index, element in enumerate(row):
                    if index not in covered_columns:
                        elements.append(element)
        # print(f'els = {len(elements)}')
        min_uncovered_num = min(elements)
        # Add m to every covered element
        adjusted_matrix = result_matrix
        for row in covered_rows:
            adjusted_matrix[row] += min_uncovered_num
        for column in covered_columns:
            adjusted_matrix[:, column] += min_uncovered_num
        # Subtract m from every element
        m_matrix = np.ones(self._shape, dtype=int) * min_uncovered_num
```



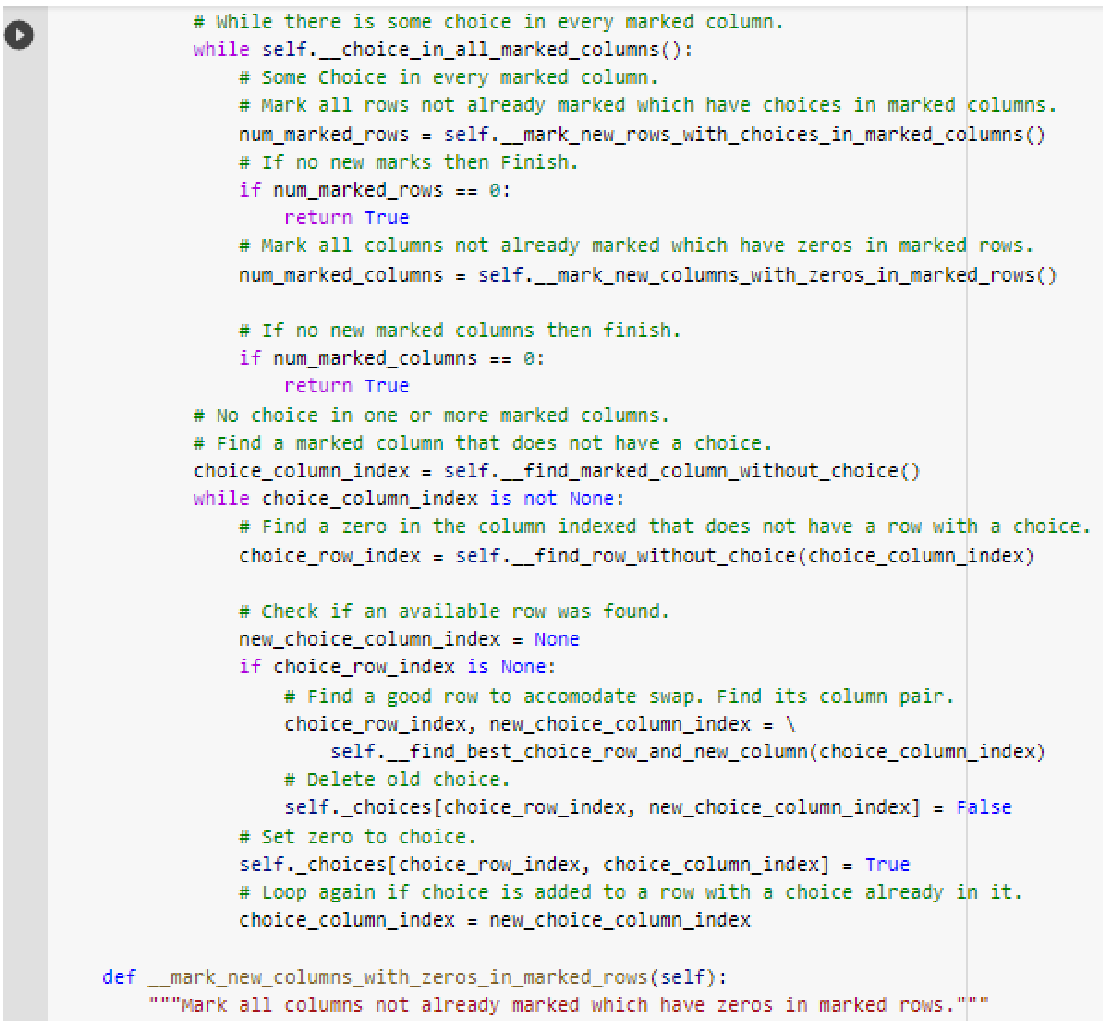
```
        adjusted_matrix -= m_matrix
        return adjusted_matrix

    def __find_matches(self, zero_locations):
        """Returns rows and columns with matches in them."""
        marked_rows = np.array([], dtype=int)
        marked_columns = np.array([], dtype=int)
        # Mark rows and columns with matches
        # Iterate over rows
        for index, row in enumerate(zero_locations):
            row_index = np.array([index])
            if np.sum(row) == 1:
                column_index, = np.where(row)
                marked_rows, marked_columns = self.__mark_rows_and_columns(marked_rows, marked_columns, row_index,
                                                                           column_index)
        # Iterate over columns
        for index, column in enumerate(zero_locations.T):
            column_index = np.array([index])
            if np.sum(column) == 1:
                row_index, = np.where(column)
                marked_rows, marked_columns = self.__mark_rows_and_columns(marked_rows, marked_columns, row_index,
                                                                           column_index)
        return marked_rows, marked_columns
    @staticmethod

    def __mark_rows_and_columns(marked_rows, marked_columns, row_index, column_index):
        """Check if column or row is marked. If not marked then mark it."""
        new_marked_rows = marked_rows
        new_marked_columns = marked_columns
        if not (marked_rows == row_index).any() and not (marked_columns == column_index).any():
            new_marked_rows = np.insert(marked_rows, len(marked_rows), row_index)
            new_marked_columns = np.insert(marked_columns, len(marked_columns), column_index)
        return new_marked_rows, new_marked_columns
    @staticmethod

    def select_arbitrary_match(zero_locations):
```



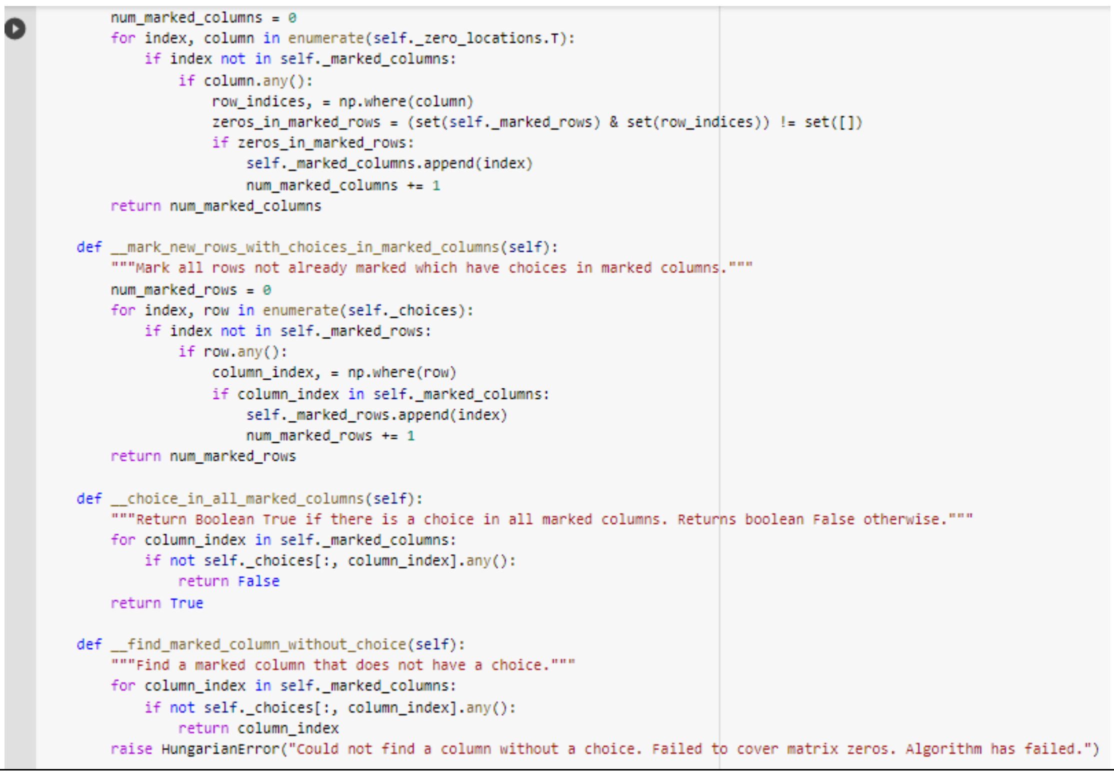
```python
    def select_arbitrary_match(zero_locations):
        """Selects row column combination with minimum number of zeros in it."""
        # Count number of zeros in row and column combinations
        rows, columns = np.where(zero_locations)
        zero_count = []
        for index, row in enumerate(rows):
            total_zeros = np.sum(zero_locations[row]) + np.sum(zero_locations[:, columns[index]])
            zero_count.append(total_zeros)
        # Get the row column combination with the minimum number of zeros.
        indices = zero_count.index(min(zero_count))
        row = np.array([rows[indices]])
        column = np.array([columns[indices]])
        return row, column

    def __set_results(self, result_lists):
        """Set results during calculation."""
        # Check if results values are out of bound from input matrix (because of matrix being padded).
        # Add results to results list.
        for result in result_lists:
            row, column = result
            if row < self._maxRow and column < self._maxColumn:
                new_result = (int(row), int(column))
                self._results.append(new_result)

class CoverZeros:
    def __init__(self, matrix):

        # Find zeros in matrix
        self._zero_locations = (matrix == 0)
        self._shape = matrix.shape

        # Choices starts without any choices made.
        self._choices = np.zeros(self._shape, dtype=bool)
        self._marked_rows = []
        self._marked_columns = []
```



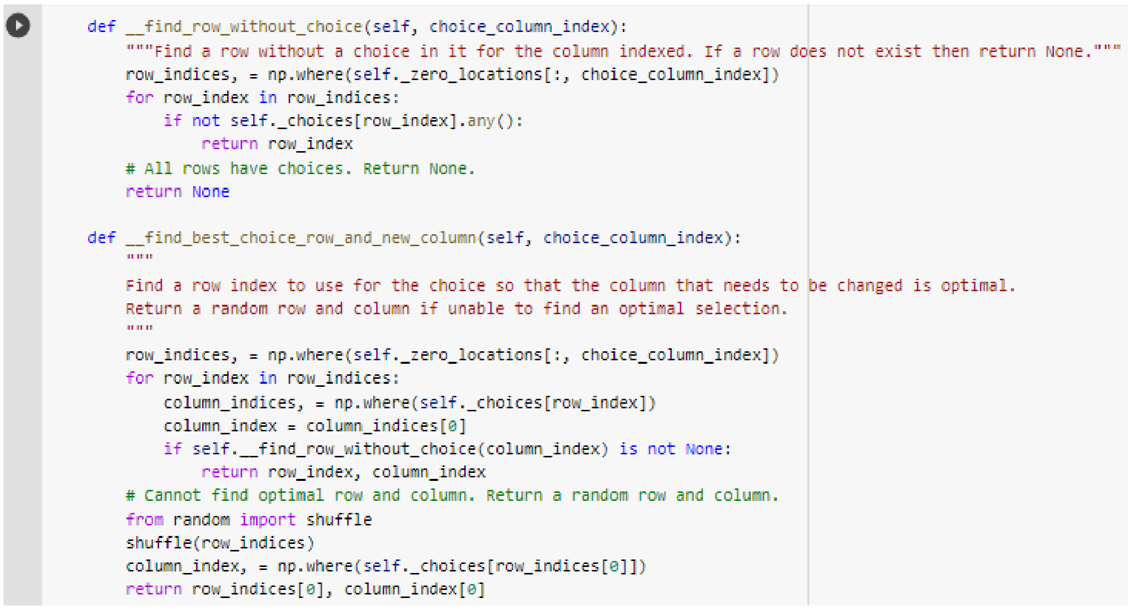
```python
        # marks rows and columns
        self.__calculate()

        # Draw lines through all unmarked rows and all marked columns.
        self._covered_rows = list(set(range(self._shape[0])) - set(self._marked_rows))
        self._covered_columns = self._marked_columns

    def get_covered_rows(self):
        """Return list of covered rows."""
        return self._covered_rows

    def get_covered_columns(self):
        """Return list of covered columns."""
        return self._covered_columns

    def __calculate(self):
        while True:
            # Erase all marks.
            self._marked_rows = []
            self._marked_columns = []

            # Mark all rows in which no choice has been made.
            for index, row in enumerate(self._choices):
                if not row.any():
                    self._marked_rows.append(index)

            # If no marked rows then finish.
            if not self._marked_rows:
                return True
            # Mark all columns not already marked which have zeros in marked rows.
            num_marked_columns = self.__mark_new_columns_with_zeros_in_marked_rows()

            # If no new marked columns then finish.
            if num_marked_columns == 0:
                return True
            # While there is some choice in every marked column.
```



```
            # While there is some choice in every marked column.
            while self.__choice_in_all_marked_columns():
                # Some Choice in every marked column.
                # Mark all rows not already marked which have choices in marked columns.
                num_marked_rows = self.__mark_new_rows_with_choices_in_marked_columns()
                # If no new marks then Finish.
                if num_marked_rows == 0:
                    return True
                # Mark all columns not already marked which have zeros in marked rows.
                num_marked_columns = self.__mark_new_columns_with_zeros_in_marked_rows()

                # If no new marked columns then finish.
                if num_marked_columns == 0:
                    return True
            # No choice in one or more marked columns.
            # Find a marked column that does not have a choice.
            choice_column_index = self.__find_marked_column_without_choice()
            while choice_column_index is not None:
                # Find a zero in the column indexed that does not have a row with a choice.
                choice_row_index = self.__find_row_without_choice(choice_column_index)

                # Check if an available row was found.
                new_choice_column_index = None
                if choice_row_index is None:
                    # Find a good row to accomodate swap. Find its column pair.
                    choice_row_index, new_choice_column_index = \
                        self.__find_best_choice_row_and_new_column(choice_column_index)
                    # Delete old choice.
                    self._choices[choice_row_index, new_choice_column_index] = False
                # Set zero to choice.
                self._choices[choice_row_index, choice_column_index] = True
                # Loop again if choice is added to a row with a choice already in it.
                choice_column_index = new_choice_column_index

    def __mark_new_columns_with_zeros_in_marked_rows(self):
        """Mark all columns not already marked which have zeros in marked rows."""
```



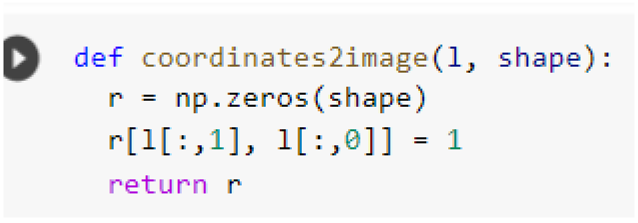
```
        num_marked_columns = 0
        for index, column in enumerate(self._zero_locations.T):
            if index not in self._marked_columns:
                if column.any():
                    row_indices, = np.where(column)
                    zeros_in_marked_rows = (set(self._marked_rows) & set(row_indices)) != set([])
                    if zeros_in_marked_rows:
                        self._marked_columns.append(index)
                        num_marked_columns += 1
        return num_marked_columns

    def __mark_new_rows_with_choices_in_marked_columns(self):
        """Mark all rows not already marked which have choices in marked columns."""
        num_marked_rows = 0
        for index, row in enumerate(self._choices):
            if index not in self._marked_rows:
                if row.any():
                    column_index, = np.where(row)
                    if column_index in self._marked_columns:
                        self._marked_rows.append(index)
                        num_marked_rows += 1
        return num_marked_rows

    def __choice_in_all_marked_columns(self):
        """Return Boolean True if there is a choice in all marked columns. Returns boolean False otherwise."""
        for column_index in self._marked_columns:
            if not self._choices[:, column_index].any():
                return False
        return True

    def __find_marked_column_without_choice(self):
        """Find a marked column that does not have a choice."""
        for column_index in self._marked_columns:
            if not self._choices[:, column_index].any():
                return column_index
        raise HungarianError("Could not find a column without a choice. Failed to cover matrix zeros. Algorithm has failed.")
```



```python
def __find_row_without_choice(self, choice_column_index):
    """Find a row without a choice in it for the column indexed. If a row does not exist then return None."""
    row_indices, = np.where(self._zero_locations[:, choice_column_index])
    for row_index in row_indices:
        if not self._choices[row_index].any():
            return row_index
    # All rows have choices. Return None.
    return None

def __find_best_choice_row_and_new_column(self, choice_column_index):
    """
    Find a row index to use for the choice so that the column that needs to be changed is optimal.
    Return a random row and column if unable to find an optimal selection.
    """
    row_indices, = np.where(self._zero_locations[:, choice_column_index])
    for row_index in row_indices:
        column_indices, = np.where(self._choices[row_index])
        column_index = column_indices[0]
        if self.__find_row_without_choice(column_index) is not None:
            return row_index, column_index
    # Cannot find optimal row and column. Return a random row and column.
    from random import shuffle
    shuffle(row_indices)
    column_index, = np.where(self._choices[row_indices[0]])
    return row_indices[0], column_index[0]
```

In this block of part, If the time of completion or the costs corresponding to every assignment is written down in a matrix form, it is referred to as a Cost matrix. The Hungarian Method is based on the principle that if a constant is added to every element of a row and/or a column of cost matrix, the optimum solution of the resulting assignment problem is the same as the original problem and vice versa. The original cost matrix can be reduced to another cost matrix by adding constants to the elements of rows and columns where the total cost or the total completion time of an assignment is zero. Since the optimum solution remains unchanged after this reduction, this assignment is also the optimum solution of the original problem. If the objective is to maximize the effectiveness through Assignment, Hungarian Method can be applied to a revised cost matrix obtained from the original matrix.

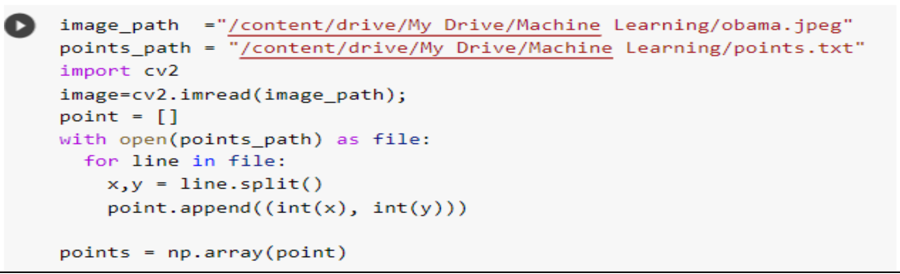
Step – 2



```python
def coordinates2image(l, shape):
    r = np.zeros(shape)
    r[l[:,1], l[:,0]] = 1
    return r
```

This block is getting the coordinates of the image under test from 2D Plane and representing them by one variable as a pixel point to retrieve info of both the coordinate points.

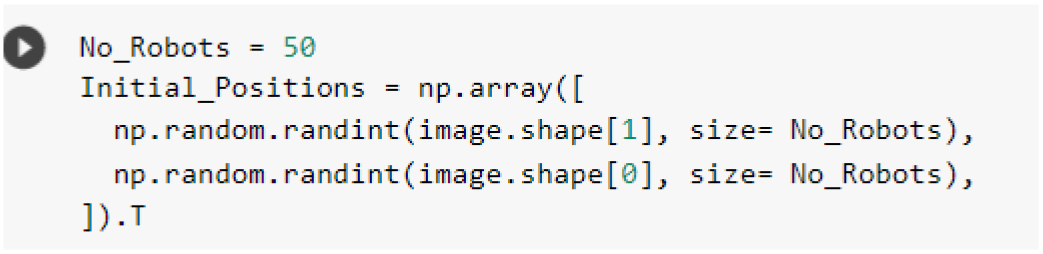
Step – 3



```python
image_path  ="/content/drive/My Drive/Machine Learning/obama.jpeg"
points_path = "/content/drive/My Drive/Machine Learning/points.txt"
import cv2
image=cv2.imread(image_path);
point = []
with open(points_path) as file:
  for line in file:
    x,y = line.split()
    point.append((int(x), int(y)))

points = np.array(point)
```

Here we are loading the image and the points from Delaunay Algorithm and storing the data in the form of x & y coordinates.

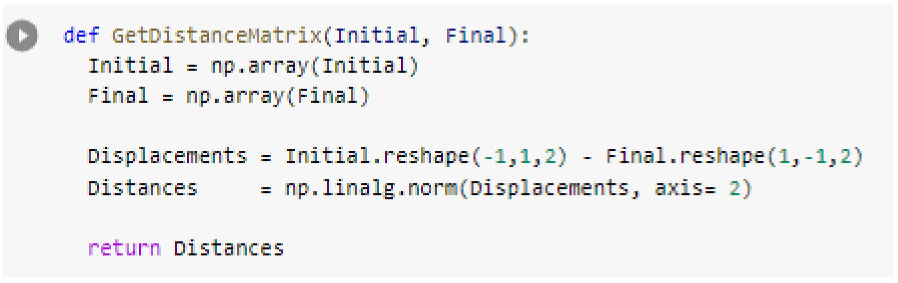
Step – 4



```
No_Robots = 50
Initial_Positions = np.array([
  np.random.randint(image.shape[1], size= No_Robots),
  np.random.randint(image.shape[0], size= No_Robots),
]).T
```

We are allocating the number of robots for the formation of patterns and randomly assigning initial positions to these robots in the form of coordinates.

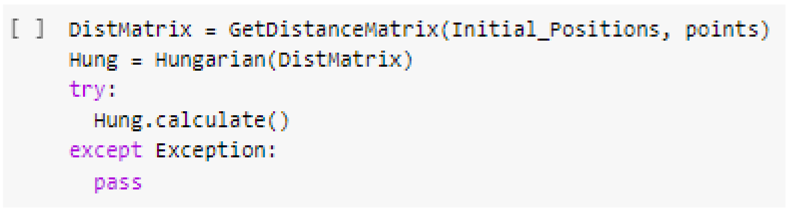
Step – 5



```
def GetDistanceMatrix(Initial, Final):
  Initial = np.array(Initial)
  Final = np.array(Final)

  Displacements = Initial.reshape(-1,1,2) - Final.reshape(1,-1,2)
  Distances     = np.linalg.norm(Displacements, axis= 2)

  return Distances
```

In this function, we are calculating the distance of the final points of the required pattern from the initial points randomly assigned to the robots in the start.

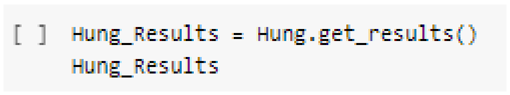
Step – 6



```
[ ]  DistMatrix = GetDistanceMatrix(Initial_Positions, points)
     Hung = Hungarian(DistMatrix)
     try:
       Hung.calculate()
     except Exception:
       pass
```

We are calling out our distance function here to give try to the points.

Step – 7



```
[ ]  Hung_Results = Hung.get_results()
     Hung_Results
```

This block of code is printing the final placement points in comparison to their initial positions.

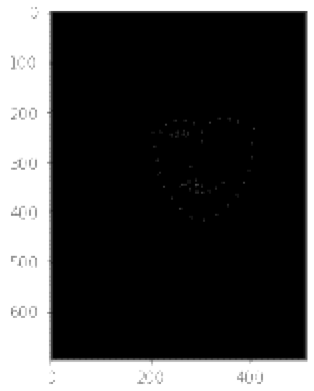
Step – 8

```
[ ] Hung_Results = np.array(Hung_Results)
    Mask = np.isin(Hung_Results[:,0], range(No_Robots))

    Final_Positions = Hung_Results[Mask][:,1]
    Final_Points    = points[Final_Positions]
```

This block of code is collecting the final placement points in the form of list and storing them.

Step – 9

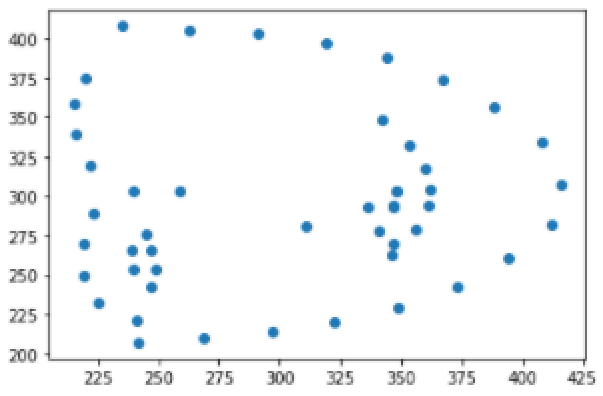
```
[ ] image_size = image.shape
    print(image_size)
    img = coordinates2image(Final_Points, image_size)

    (697, 512, 3)

[ ] plt.imshow(img, cmap= 'gray')

    <matplotlib.image.AxesImage at 0x7f676854bf10>
```



Printing out the gray points on black scale to provide a map of placement points of robots over a dark background.
(look closely, points are there but hard to see due to small dimension)

Step – 10

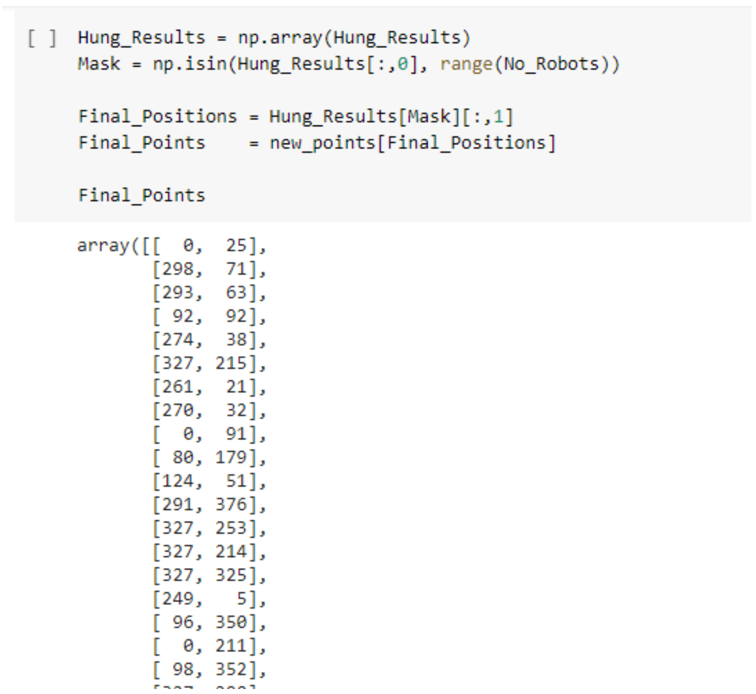
```
[ ] plt.plot(Final_Points[:,1], Final_Points[:,0], 'o')

[<matplotlib.lines.Line2D at 0x7f67684b1a10>]
```



Finally getting the possible required formation points according to the robot resources.

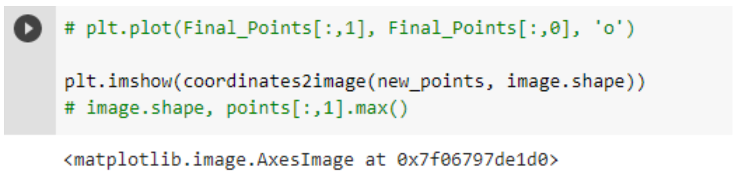
Results of Hungarian

Test 1: Octagon Shape

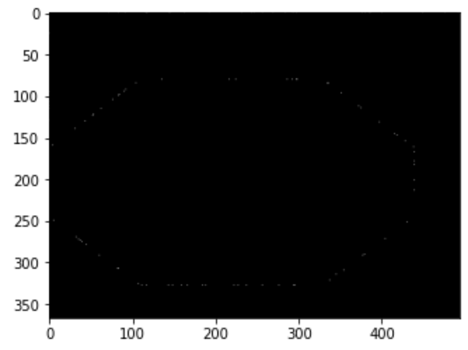


```
[ ]  Hung_Results = np.array(Hung_Results)
     Mask = np.isin(Hung_Results[:,0], range(No_Robots))

     Final_Positions = Hung_Results[Mask][:,1]
     Final_Points    = new_points[Final_Positions]

     Final_Points

     array([[  0,  25],
            [298,  71],
            [293,  63],
            [ 92,  92],
            [274,  38],
            [327, 215],
            [261,  21],
            [270,  32],
            [  0,  91],
            [ 80, 179],
            [124,  51],
            [291, 376],
            [327, 253],
            [327, 214],
            [327, 325],
            [249,   5],
            [ 96, 350],
            [  0, 211],
            [ 98, 352],
```

New final positions are assigned that are plotted below.

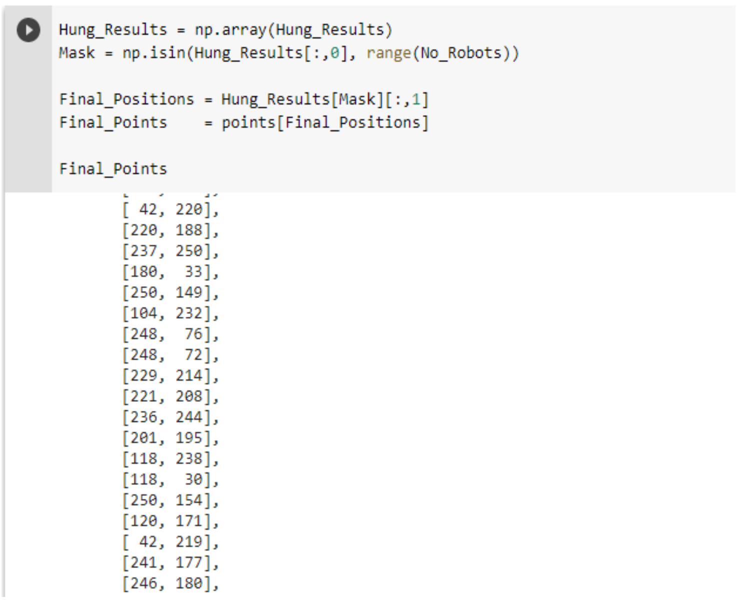


```
# plt.plot(Final_Points[:,1], Final_Points[:,0], 'o')

plt.imshow(coordinates2image(new_points, image.shape))
# image.shape, points[:,1].max()

<matplotlib.image.AxesImage at 0x7f06797de1d0>
```



Optimal Assignment

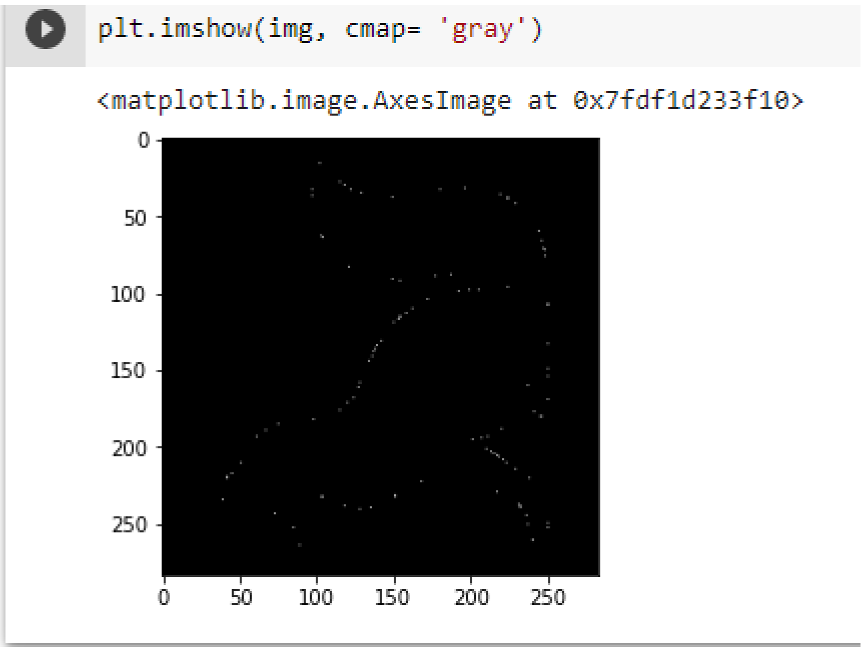
Test 2: <u>FOX Image</u>



```
Hung_Results = np.array(Hung_Results)
Mask = np.isin(Hung_Results[:,0], range(No_Robots))

Final_Positions = Hung_Results[Mask][:,1]
Final_Points    = points[Final_Positions]

Final_Points
```

```
       [ 42, 220],
       [220, 188],
       [237, 250],
       [180,  33],
       [250, 149],
       [104, 232],
       [248,  76],
       [248,  72],
       [229, 214],
       [221, 208],
       [236, 244],
       [201, 195],
       [118, 238],
       [118,  30],
       [250, 154],
       [120, 171],
       [ 42, 219],
       [241, 177],
       [246, 180],
```

New final positions are assigned that are plotted below.



```
plt.imshow(img, cmap= 'gray')
```

```
<matplotlib.image.AxesImage at 0x7fdf1d233f10>
```

Optimal Assignment

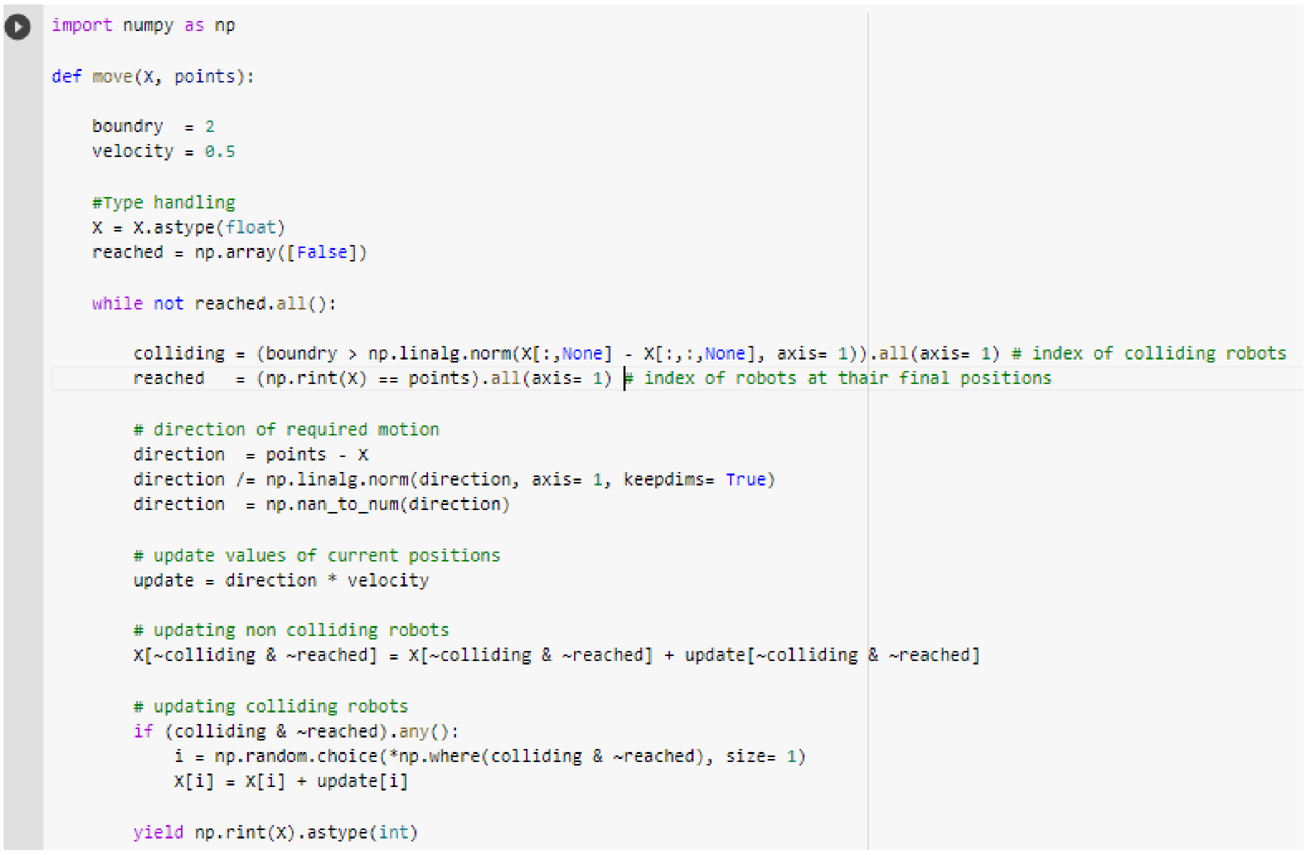
3. *<u>Collision Avoidance</u>*

- Finally, to ensure visually appealing smooth motions and collision-free trajectories **Distributed Collision Avoidance Method** will be used.

  Brief – We ultimately visualize that the commanded multi-robot system to the respective destination might undergo some collision in their trajectory throughout their journey as not being intelligent enough to automatically detect and be able to handle

by itself. Here in this part, we provided handling for such a disturbance which might result in disrupting our predicted pattern and damaging of hardware as well.
We in our code provided a check for all our robots according to the resources that if there is a collision in path of any robot with any other, one of the two colliding robots stops its movement while the other continues. There is an arbitrary boundary considered to be breached every time we say a collision is detected. Here in our case, we have opted the boundary to be 2 pixels. As soon as one of the colliding bodies achieve some distance, the stopped robot starts moving to the destination point again and ultimately the collision free trajectories are achieved.

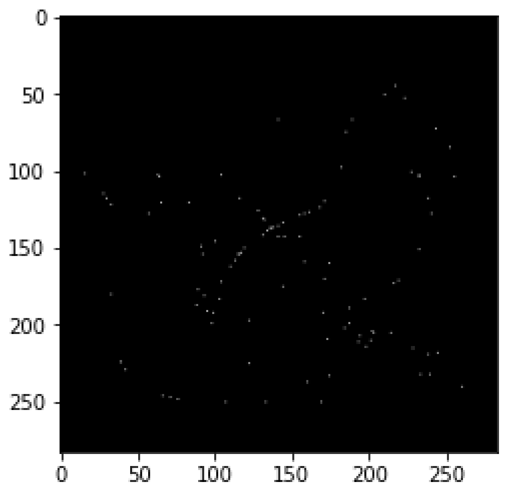
CODE



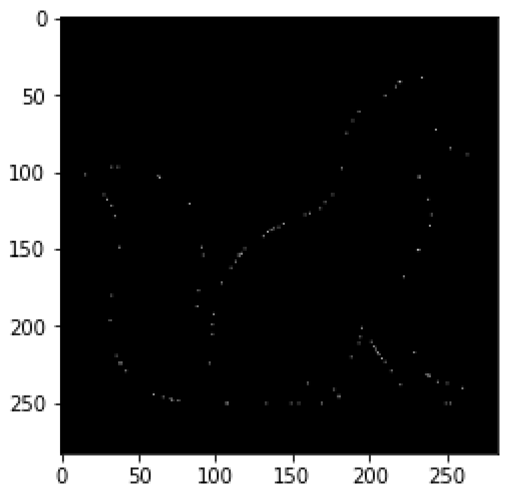
```python
import numpy as np

def move(X, points):

    boundry  = 2
    velocity = 0.5

    #Type handling
    X = X.astype(float)
    reached = np.array([False])

    while not reached.all():

        colliding = (boundry > np.linalg.norm(X[:,None] - X[:,:,None], axis= 1)).all(axis= 1) # index of colliding robots
        reached   = (np.rint(X) == points).all(axis= 1) # index of robots at thair final positions

        # direction of required motion
        direction  = points - X
        direction /= np.linalg.norm(direction, axis= 1, keepdims= True)
        direction  = np.nan_to_num(direction)

        # update values of current positions
        update = direction * velocity

        # updating non colliding robots
        X[~colliding & ~reached] = X[~colliding & ~reached] + update[~colliding & ~reached]

        # updating colliding robots
        if (colliding & ~reached).any():
            i = np.random.choice(*np.where(colliding & ~reached), size= 1)
            X[i] = X[i] + update[i]

        yield np.rint(X).astype(int)
```

OUTPUT

Test 1: FOX Image



Unarranged (In process to approach the final destination)

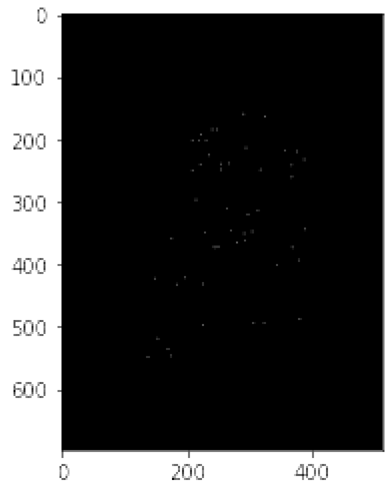


Arranged(After Avoiding all Collisions and Exceptions)

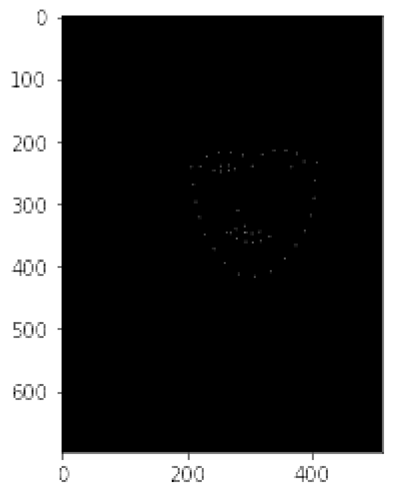
The video demo for Fox Picture Formation is as follows:

https://drive.google.com/file/d/1JnG_ksRrNoPEauc_wCkJPfa_dX5dXRPD/view?usp=sharing

Test 2: OBAMA's Image



Unarranged (In process to approach the final destination)



Arranged(After Avoiding all Collisions and Exceptions)

- We defined the arbitrary boundary limit as 2 pixels per robot and likewise velocity with which our robot would move to the destination.
- There's this type handling of integer type points to the float type so that minimal deviation is noticed as well.
- We are running a loop eventually which keeps updating the current positions based on the collision detection.
- Inside loop, we initially are checking whether the distance between any of the robot is less than the assumed boundary or not. Secondly, we are checking if the random initial placements of the robots are already at the destination or not.
- Further we are estimating the direction of required motion of every robot which is to move to the destination point.

- We get a unit vector in the required direction for every robot, and we multiply this vector with velocity for each robot to traverse the robot translationally.
- We check that if the robots have not arrived yet and are colliding, we take both of the robots and only traverse one of the two robots at a time until the robots have excluded out of each other's boundary.
- This loop works until all the robots at their initial positions have updated on their final positions.
- The final required pattern is obtained with collision free trajectories of Robots.

Conclusion

In this project we generated the points of image or a given shape using Canny Detection and generated the optimal pattern using the Delaunay and Voronoi Algorithms. The problems we faced were while using shapes, as the points got placed in the internal space rather than the boundary of the shape. This problem is completely resolved now using Canny Edge Algorithm, that does a good job in detection.
Then we implemented the Hungarian Algorithm for the optimal assignments of the robots so that each robot is assigned a final position that is closest to that. The problems we faced were having different errors while using lists and matrices for data, so we converted all the data into arrays using the NumPy library. The next step is a simpler one. We will implement the Collision Detection Algorithm which will consider the paths and the velocities of the neighboring robots and will formulate a collision-free trajectory. Although the approaching time might increase but the good part of it is that it is not at the cost of efficient and desirable output. It was a great experience of learning and implementing these powerful algorithms.