Weighted Centroid Computation and Different Boundaries Shapes

Bruno Maciel
University of California, San Diego
Summer 2015

1. Summary

This report covers the c++ codes written to compute the Voronoi cells' centroids, using different boundaries shapes, as rectangle, ellipsoid, and some "random" polygon. All images provided in this report were obtained by testing in MATLAB the result taken from the code written in c++.

2. Big Picture

The goal in Dr. Costes's and Dr. Martinez's labs is to implement formation control, making the turtlebots and airdrones able to explore the space avoiding obstacles and collisions. Partitioning the plane in Voronoi cells and computing each cell's centroids make the robots dispose themselves equally on the plane, while the weighted centroid allow to define which part of the plane is more important to explore, making the robots move to and surround that area.

3. My Contributions

During the Summer Quarter I worked with Julio Martinez. Most part of my role was writing and optimizing c++ codes, plotting the results on MATLAB and correcting errors. The Fortune's Algorithm split the space into voronoi cells, after that, the algorithm we developed computes the centroids of those cells, overcomes the Fortune's Algorithm boundary shape limitation and allow us to define a density function which states the most important part of the plane, which makes the robots surround that area.

4. Methodology

4.1 Computing the Centroid

Julio and me worked together to write the code that computes the centroids, which became the foundation of all further code we wrote.

The first piece of the code is the Fortune's Algorithm, which returns the beginning and ending point of each line of each voronoi cell. So, first, is necessary to store the position of each point (vertex) and filter the repeated ones in order to avoid unnecessary further calculation.

Second step, enumerate the vertices and find out which vertices compound which cell. After that, for each cell, vertices are ordered in a counterclockwise way based on the angle between the vertices and the site.

Finally, the centroids are computed using a formula based on the area of each cell [1].

```
Cell 1:
Nº Vertices = 4
Vertices    = 9, 1, 7, 5

Cell 2:
Nº Vertices = 5
Vertices    = 5, 7, 8, 6, 4

Cell 3:
Nº Vertices = 4
Vertices    = 4, 6, 3, 12

Cell 4:
Nº Vertices = 5
Vertices    = 6, 8, 2, 11, 3

Cell 5:
Nº Vertices = 5
Vertices    = 1, 10, 2, 8, 7
```

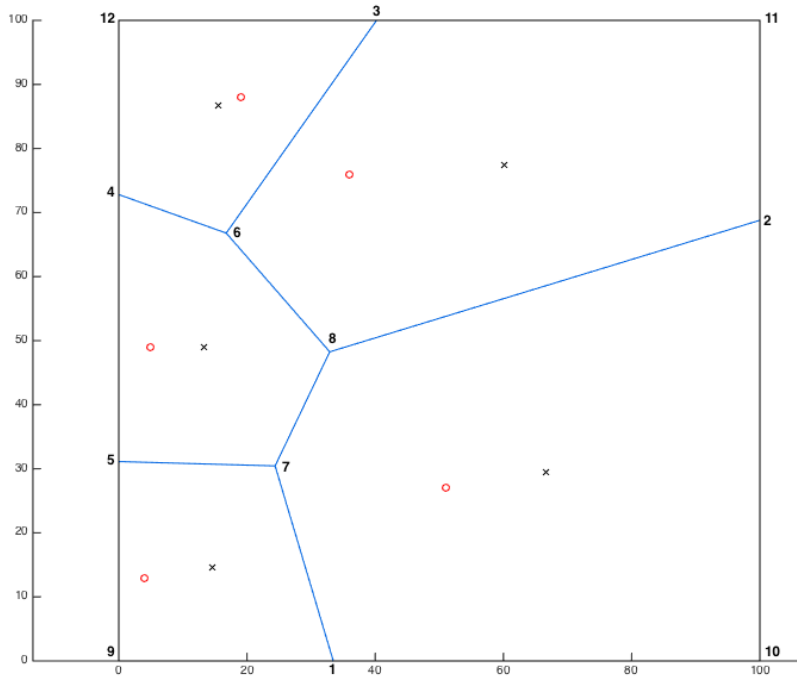


Figure 1. Red circles represents the sites (robots present position). Black 'x' represents the computed centroid (target to move)

This process is repeated many times, until the actual position of the robots (sites) converges with the centroids' position.

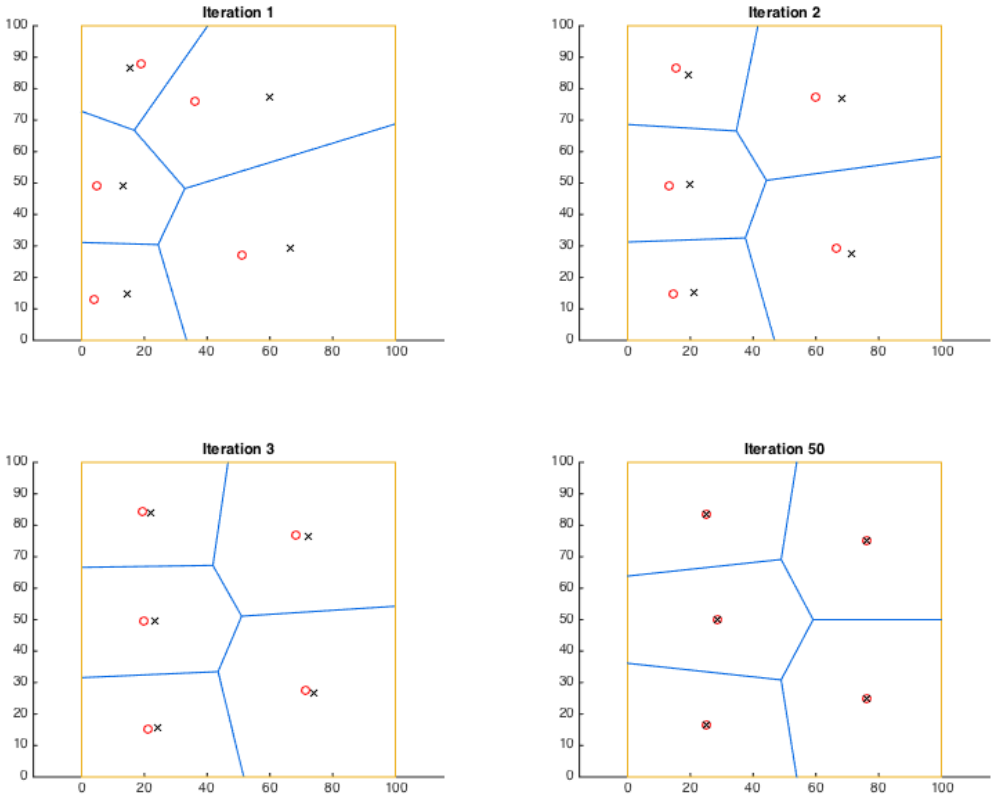


Figure 2. The centroids computed in the previous iteration are used as the current sites and a new centroid is computed

4.2 Boundaries

Originally any rectangle was used as plane for the Voronoi diagram because Fortune's Algorithm requires the edges of a rectangle as an input. However, the necessity for using other shapes as boundary arose.

4.2.1 Ellipsoid Boundary

As it is still need to provide a rectangle for the Fortune's Algorithm, the best way to make an Ellipsoid as boundary is fitting the biggest rectangle as possible inside it. So, it is needed to provide the center position of the ellipsoid and the X-radius and Y-radius, after that the edges of the rectangle are calculated.

Let's call the vertices (1, 2, 3, 4 and 5) as the "outside vertices" as they are at the boundary of the plane. Fortune's Algorithm always return the position of the outside vertices at the yellow rectangle. So, it is necessary to find out where those vertices should be at the ellipse boundary. To explain how it is done, I will take as an example the vertices 6 and 3:

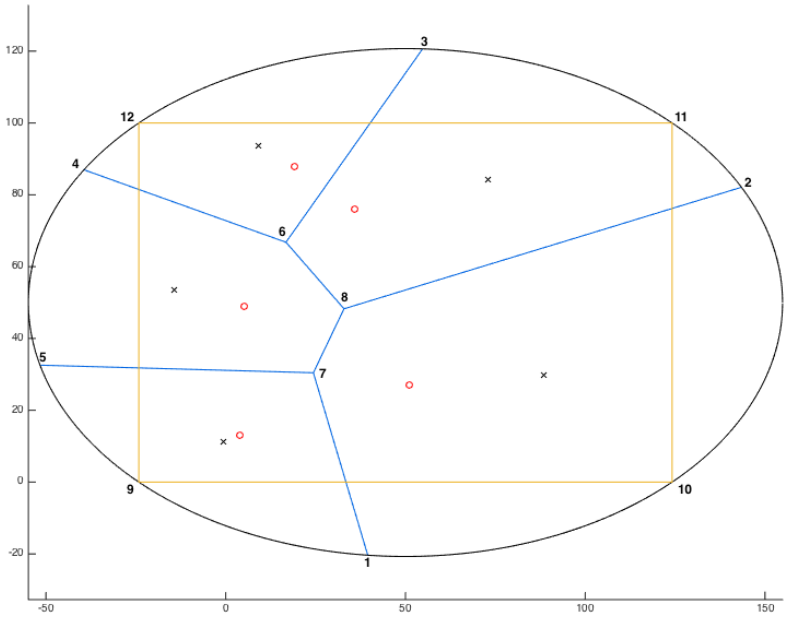


Figure 3. Fitting the biggest rectangle in an ellipse.

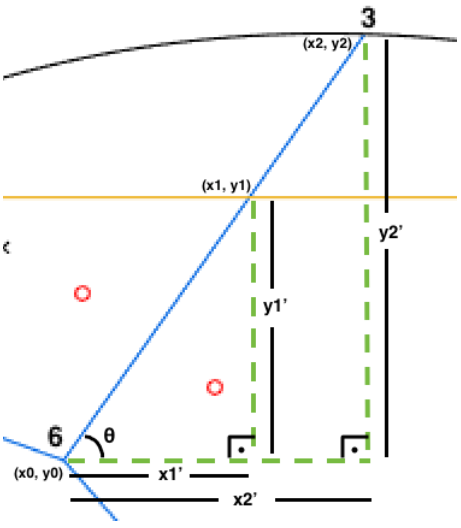
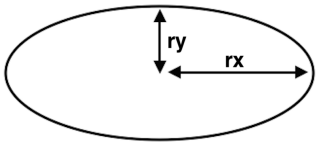


Figure 4.

The following equation describes an ellipse:



$$\frac{(x - xc)^2}{rx^2} + \frac{(y - yc)^2}{ry^2} = 1$$

We can define x2 = x0 + x2' and y2 = y0 + y2'.

We also can state that x2' = k * x1' and y2' = m * y1', being k and m scalars.

Vertice (x1, y1) is given by Fortune's Algorithm as well as (x0, y0). We know that x1' = x1 - x0 and y1' = y1 - y0.

So, x2 = x0 + k(x1 - x0) and y2 = y0 + m(y1 - y0).

As tan θ = y1'/x1' = y2'/x2'. We get k = m.

Now, just plug x2 and y2 in the ellipse equation and we'll get the value of k, and consequently x2 and y2.

Finally, the vertices (x2, y2) are stored in replacement of (x1, y1).

However, an issue arise, the centroid formula does not take in account curves, only straight lines. So in order to get a better approximated centroid, it is added 4 more vertices (vertices 13 to 16) to the calculations.

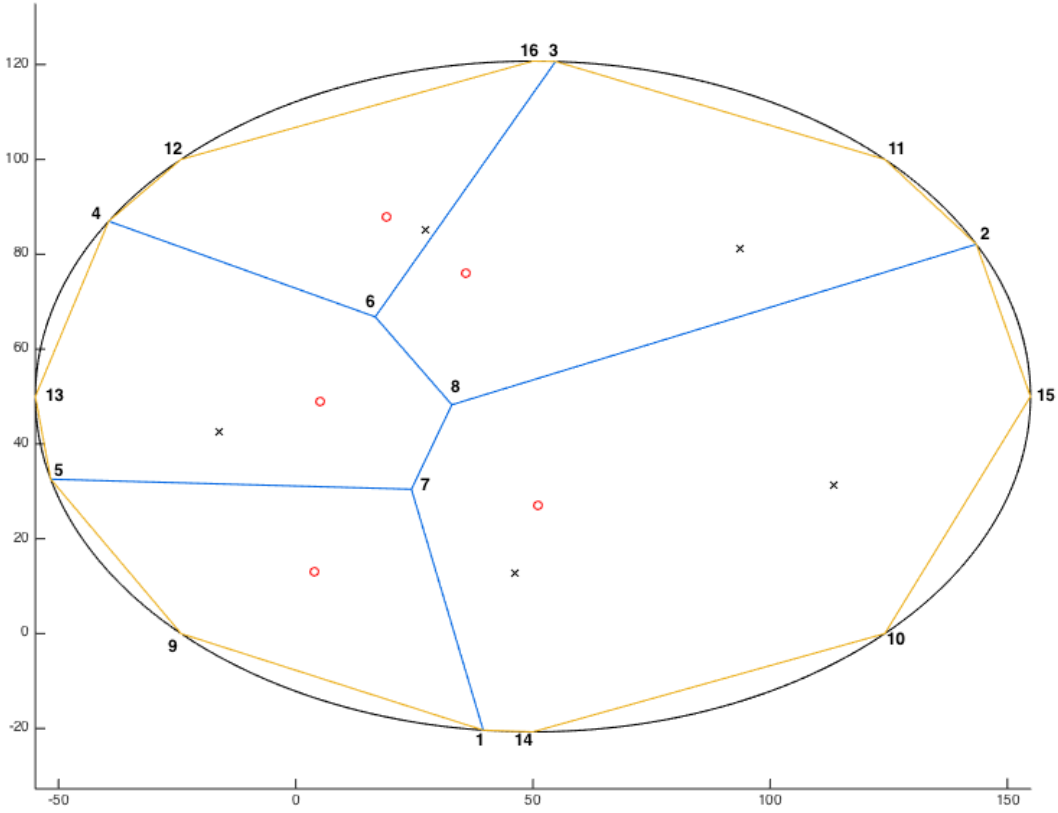


Figure 5. The areas between the yellow polygon and the black ellipse are not taken in account to compute the centroid. As those areas are relatively small, the result is a good approximation.

4.2.2 "Crazy" Boundary

Now, for a polygon boundary, finding the vertices at the boundary follows the same idea of the ellipse boundary. However, there's no function that describe the whole boundary, but each boundary line will have it's own function.

The code looks for vertices at the rectangle boundary, once it finds them, it will place those vertices at the polygon boundary.

To explain how the code works, let's take as an example the vertex 4.

The boundary is compounded by 8 vertices (numbers 9 to 16), let's call them bvertices. First, the code finds the bvertex n which is closest to the vertex 4, which will be number 15. After that, the code finds the function that describe the line which contains the vertices 4 and 6, as well as the functions that describe the lines which contain the bvertices n and n-1 and the bvertices n and n+1. So, now we have the function that describes the lines 6-4, 14-15 and 15-16.

After that, we find the intersection between the 6-4 line and the others two lines, compare the distance from those intersection points to the vertex 6, and discard the farthest point.

Finally, replace those original vertices by the new ones.

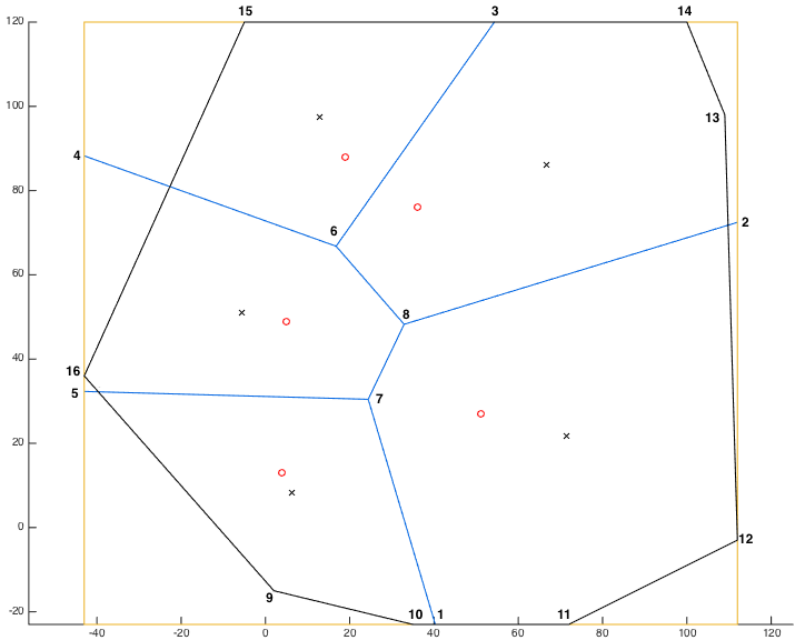


Figure 6. Initial vertices taken from Fortune's Algorithm

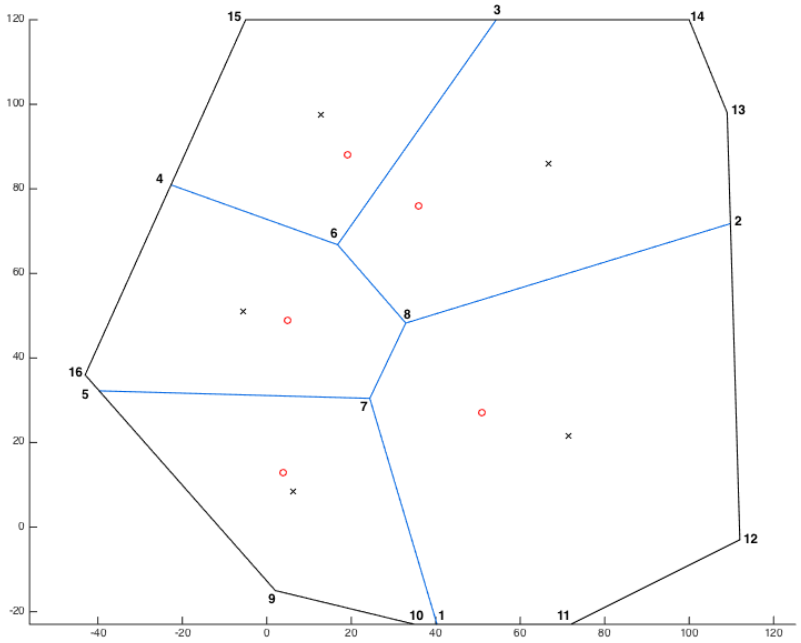


Figure 7. After calling the function that sets the vertices on the polygon boundary

5. Tips and Pitfalls

Along the code, I left many print statements in the form of comment. In case of having some error, uncommenting those lines can help find where is exactly the error happens.

One of the most important functions in the code is the *truncate* function. As the code works with floating point values, it is essential to define the number of digits after the decimal point we will work with. Throughout the code two values are compared in order to define if they are equal, smaller or grater than

the other, that's the moment where the truncate function is called making two values that only differ from each other at the i^{th} decimal place equal if i is greater than the value passed in the truncate function.

By testing, the largest number for i I got that makes the code works with no errors is 3, however it is possible that in future tests, this number must be changed to 2. One trick that helps to not reduce this value below 3 is the error variable E in the *getIndices* function. In case of future necessity to modify those values, I would recommend to change first the error variable E.

```
float CentroidGenerator::truncate(float num, int precision)
{
    num = num*pow(10,precision);
    num = round(num);
    num = num/pow(10,precision);
    return num;
}

float CentroidGenerator::distance(float siteX, float siteY, float vertX, float vertY)
{
    float dist = sqrt(pow((vertY-siteY),2) + pow((vertX-siteX),2));
    dist = truncate(dist,3);
    return dist;
}

void CentroidGenerator::getIndices(Matrix indicesMatrix, Matrix Sites, int nSites, int
nVertices)
{
    const float E = 0.002;

    //find out which vertice is part of which cell
    for (int i=1;i<nSites;i++) {
        for (int j=1;j<=nVertices;j++) {
            if (indicesMatrix.elements[i-1][j-1]<0) {
                int p=-1;
                for (int k=i;k<nSites;k++){
                    float a = distance(Sites.elements[i-1][0], Sites.elements[i-1][1],
uniqueVertices.elements[j-1][0], uniqueVertices.elements[j-1][1]);
                    float b = distance(Sites.elements[k][0], Sites.elements[k][1],
uniqueVertices.elements[j-1][0], uniqueVertices.elements[j-1][1]);
                    if (a < b-E)
                    {   indicesMatrix.setElement(k, j-1, 0);  }
                    else if (a <= b+E && a >= b-E)
                    {
                        indicesMatrix.setElement(i-1, j-1, j);
                        indicesMatrix.setElement(k, j-1, j);
                        p=k;
                    }
                    else
                    {
                        indicesMatrix.setElement(i-1, j-1, 0);
                        if (p>0) indicesMatrix.setElement(p, j-1, 0);
                        k=nSites+1;
                    }
                }
            }
        }
    }
    for (int i=0; i<nSites; i++)
    {
        for (int j=0; j<nVertices; j++)
        {   if (indicesMatrix.elements[i][j]<0) indicesMatrix.setElement(i, j, j+1); }
    }

    //indicesMatrix.printArray("Indices");
}
```

The “crazy” boundary code was made so that someone can draw some random polygon on the table, defining the area the robots should occupy. But, one important thing I must make it clear is that, the code does not work for any random polygon. To make it works, is necessary to make the polygon a convex hull, there are many algorithms able to do that, but the Graham Scan [2] seems to be very efficient for our purpose. So, before the drawn polygon be sent to the code, first is necessary to pass it through the convex hull algorithm, and after that the code will work properly.

6. Conclusion and Future Work

The c++ code that computes the centroid of the voronoi cells as well as the functions to work with different boundaries shapes seems to be very efficient and fast, even though the formula to compute the centroid does not take in account curve boundaries, but only straight lines, making the computation of the centroid in a ellipsoid boundary shape just an approximation.

Me and Julio, to replace the centroid function, we started working on a function that computes the center of mass of the voronoi cells, defining a density function, which can make the centers of mass move towards wherever the density function has it’s highest value. Working to improve the function to compute the center of mass, I believe, is the next step for this project.

References

[1] Paul Bourke. Calculating the Area and Centroid of a Polygon, 1988.
[2] Graham Scan. https://en.wikipedia.org/wiki/Graham_scan

Referenced Code
CentroidGenerator_CzyBdy.cpp

```
#include <iostream>
#include <stdio.h>
#include <search.h>
#include <stdlib.h>
#include <math.h>
#include <vector>
#include "CentroidGenerator_CzyBdy.h"
#include "VoronoiDiagramGenerator.h"
using namespace std;

Matrix uniqueVertices(50,2);

Matrix::Matrix(int r, int c, float data) {
    rows = r;
    cols = c;
    elements= new float*[rows];
    for (int i=0; i<rows; i++)
    {
        elements[i] = new float[cols];
        for (int j=0; j < cols; j++)    {   elements[i][j]=data;    }
    }
}
Matrix::~Matrix(){
    //cout << endl << "End program" << endl;
}
void Matrix::setMatrix(int r, int c, float data) {
    elements = new float*[r];
    for (int i=0; i<r; i++)
    {
        elements[i] = new float[c];
        for (int j=0; j<c; j++)
```

```
        {   elements[i][j]=data;  }
    }
    setRows(r);
    setCols(c);
}
void Matrix::printArray(string name) {
    cout << name << ": " << endl;
    for(int i=0;i<rows;i++) {
        for(int j=0;j<cols;j++) {   cout << elements[i][j] << "\t";   }
        cout << endl;
    }
    cout << endl;
}
int Matrix::setRows(int r) {
    rows = r;
    return rows;
}
int Matrix::setCols(int c) {
    cols = c;
    return cols;
}
void Matrix::setElement(int r, int c, float data){
    elements[r][c]=data;
}


float CentroidGenerator::truncate(float num, int precision)        //truncate a
floating point number
{   num = num*pow(10,precision);
    num = round(num);
    num = num/pow(10,precision);
    return num;
}

float CentroidGenerator::distance(float siteX, float siteY, float vertX, float vertY)
//calculate the distance between a site and a vertice
{   float dist = sqrt(pow((vertY-siteY),2)+pow((vertX-siteX),2));
    dist = truncate(dist,3);        //using 4 as precision was identified some errors
in calculations. So use 3 or less as precision
    return dist;
}

CentroidGenerator::~CentroidGenerator(){}

void CentroidGenerator::getUniqVertices(std::vector<float> allVerticesVector)
{
    for (int i=0; i<allVerticesVector.size(); i++)  //as numbers are float, need to
truncate them in order to work
    {   allVerticesVector[i]=truncate(allVerticesVector[i], 4); }

    Matrix allVerticesMatrix(50,2); //initialize the matrix with 50 rows, but it will
be replaced by the next line. So instead of 50, it can be any number
    allVerticesMatrix.rows = allVerticesVector.size()/2;

    //transform vector "allVerticesVector into a matrix
    int k=0;
    for (int i=0;i<allVerticesMatrix.rows;i++)
    {
        for (int j=0;j<2;j++)
{    allVerticesMatrix.setElement(i,j,allVerticesVector[k]); k++;    }
    }
    //allVerticesMatrix.printArray("All Vertices");
```

```
    std::vector<int> repeatIndex={};    //vector to store the indices(rows) of
repeated vertices

    //find indices(rows) that repeats its elements == vertices that appear more than
once
    for (int i=0; i<allVerticesMatrix.rows; i++){
        for (int j=i+1; j<allVerticesMatrix.rows; j++) {
            if (allVerticesMatrix.elements[i][0] == allVerticesMatrix.elements[j][0]
&& allVerticesMatrix.elements[i][1] == allVerticesMatrix.elements[j][1])
            {
                repeatIndex.push_back(i);       //if elements of a row repeats, add
its index in the vector
                j=allVerticesMatrix.rows;       //break loop once it's found,
otherwise if it repeats more than once, it'll store same index twice
            }
        }
    }

    //store non repeated vertices in the "uniqueVertices" matrix
    uniqueVertices.setRows(allVerticesMatrix.rows-repeatIndex.size());
    int a=0;
    for (int i=0; i<allVerticesMatrix.rows; i++) {
        int count=0;
        for (int j=0; j<repeatIndex.size(); j++) //make sure the index i is different
from all indices storage in repeatIndex
        {
            if (i!=repeatIndex[j])  count++;
        }
        if (count==repeatIndex.size())   //if index i isn't storage in repeatIndex: we
can storage the values of row (i) from Vertices to newVertices
        {
            uniqueVertices.setElement(a, 0, allVerticesMatrix.elements[i][0]);
            uniqueVertices.setElement(a, 1, allVerticesMatrix.elements[i][1]);
            a++;
        }
    }
    uniqueVertices.printArray("Unique Vertices");
}

void CentroidGenerator::getIndices(Matrix indicesMatrix, Matrix Sites, int nSites, int
nVertices)
{
    const float E = 0.002;
    //find out which vertice is part of which cell
    for (int i=1;i<nSites;i++) {
        for (int j=1;j<=nVertices;j++) {               //go through each vertice to
find what site is nearer to that vertice
            if (indicesMatrix.elements[i-1][j-1]<0) {  //check if element of matrix
Indices is still with initial value -1, avoiding unecessary calculations
                int p=-1;
                for (int k=i;k<nSites;k++){            //compare the site (i) to every
other site
                    float a = distance(Sites.elements[i-1][0], Sites.elements[i-1][1],
uniqueVertices.elements[j-1][0], uniqueVertices.elements[j-1][1]);
                    float b = distance(Sites.elements[k][0], Sites.elements[k][1],
uniqueVertices.elements[j-1][0], uniqueVertices.elements[j-1][1]);
                    if (a < b-E)    //if distance from site (i) to vertice (j) is
smaller than from site (k) to vertice (j): vertice is not part of cell (k)
                    {   indicesMatrix.setElement(k, j-1, 0);  }
                    else if (a <= b+E && a >= b-E)  //if distances between two sites
and a vertice are equal: vertice is part of both cells
                    {
                        indicesMatrix.setElement(i-1, j-1, j);
                        indicesMatrix.setElement(k, j-1, j);
```

```
                        p=k;    //store the value of indice (k), in case that the
distance is equal but the cell is not the closest to the vertice (j)
                    }
                    else        //if distance from site (i) to vertice (j) is bigger
than from site (k): vertice is not part of cell (i)
                    {
                        indicesMatrix.setElement(i-1, j-1, 0);
                        if (p>0) indicesMatrix.setElement(p, j-1, 0);   //if two sites
have same distance to a vertice, but their not the closest to that vertice, it is used
to correct that failure
                        k=nSites+1; //break loop, otherwise it will go back and
compare the site (i) again with other sites, which is unnecessary
                    }
                }
            }
        }
    }
    for (int i=0; i<nSites; i++)    //some elements will still have the value -1. It
means that those vertices are actually the plane's edges.
    {
        for (int j=0; j<nVertices; j++)
        {   if (indicesMatrix.elements[i][j]<0) indicesMatrix.setElement(i, j, j+1); }
    }                           //(they're only part of one single cell)

    //indicesMatrix.printArray("Indices");
}

void CentroidGenerator::findVerticesOnCrazyBdy(float x1, float y1, float x0, float y0,
int row, int nCzyBdyVert)
{
    float m1 = (y1-y0)/(x1-x0);
    float b1 = -x1*m1+y1;
    float a1, a2;
    int index;

    for (int i=uniqueVertices.rows-nCzyBdyVert; i<uniqueVertices.rows;i++)  //find
which boundary vertice is nearer to the vertice at the boundary
    {
        if (i==uniqueVertices.rows-nCzyBdyVert) {
            a1 = distance(x1,y1,uniqueVertices.elements[i]
[0],uniqueVertices.elements[i][1]);
            index = i;
        }
        else {
            a2 = distance(x1,y1,uniqueVertices.elements[i]
[0],uniqueVertices.elements[i][1]);
            if (a1>a2) {
                a1=a2;
                index = i;
            }
        }
    }

    float x2 = uniqueVertices.elements[index][0], y2 = uniqueVertices.elements[index]
[1];

    float x3, y3, x4, y4;
    //cout << "aa: " << index << endl;
    if (index==uniqueVertices.rows-nCzyBdyVert) {
        x3 = uniqueVertices.elements[uniqueVertices.rows-1][0], y3 =
uniqueVertices.elements[uniqueVertices.rows-1][1];
    }
    else x3 = uniqueVertices.elements[index-1][0], y3 =
uniqueVertices.elements[index-1][1];
```

```
    if (index==uniqueVertices.rows-1) {
        x4 = uniqueVertices.elements[uniqueVertices.rows-nCzyBdyVert][0], y4 =
uniqueVertices.elements[uniqueVertices.rows-nCzyBdyVert][1];
    }
    else x4 = uniqueVertices.elements[index+1][0], y4 = uniqueVertices.elements[index
+1][1];
    //cout << "test: " << x4 << ", " << y4 << endl;

    float m4 = (y4-y2)/(x4-x2);
    float b4 = -x2*m4+y2;
    float m3 = (y3-y2)/(x3-x2);
    float b3 = -x2*m3+y2;

    float xp = (b4-b1)/(m1-m4), yp = m1*xp+b1;
    float xpp = (b3-b1)/(m1-m3), ypp = m1*xpp+b1;
    xp = truncate(xp, 4); yp = truncate(yp, 4);
    xpp = truncate(xpp, 4); ypp = truncate(ypp, 4);

    if (distance(xp, yp, x0, y0) > distance(xpp, ypp, x0, y0)) {
        uniqueVertices.elements[row][0]=xpp;
        uniqueVertices.elements[row][1]=ypp;
    }
    else {
        uniqueVertices.elements[row][0]=xp;
        uniqueVertices.elements[row][1]=yp;
    }

}

void CentroidGenerator::setVerticesOnBoundary(Matrix indicesMatrix, float minX, float
maxX, float minY, float maxY, int nCzyBdyVert)
{
    for (int i=0; i<uniqueVertices.rows-nCzyBdyVert; i++)     //ignore the last
nCzyBdyvert rows, because they're the edges of the CrazyBoundary
    {
        if (uniqueVertices.elements[i][0]==minX || uniqueVertices.elements[i][0]==maxX
|| uniqueVertices.elements[i][1]==minY || uniqueVertices.elements[i][1]==maxY)   //
find if vertice is at the boundary
        {
            float x1 = uniqueVertices.elements[i][0], y1 = uniqueVertices.elements[i]
[1];
            int myvector[2] = {-1,-1};
            int j=0;
            for (int k=0; k<indicesMatrix.rows; k++)    //store the indices of the
cells, which that vertice is part of
            {
                if (indicesMatrix.elements[k][i] == i+1)    {
                    myvector[j] = k;
                    j++;
                }
            }
            //cout << "abc: " << myvector[0] << ", " << myvector[1] << endl;
            for (int k=0; k<indicesMatrix.cols-nCzyBdyVert; k++)
            {
                if (indicesMatrix.elements[myvector[0]][k] ==
indicesMatrix.elements[myvector[1]][k] && indicesMatrix.elements[myvector[0]][k] != i
+1 && indicesMatrix.elements[myvector[0]][k] > 0)    //find the "internal" vertice
that is also part of the same two cells
                {
                    float x0 = uniqueVertices.elements[k][0], y0 =
uniqueVertices.elements[k][1];
                    findVerticesOnCrazyBdy(x1, y1, x0, y0, i, nCzyBdyVert);
                    k=indicesMatrix.cols;   //break the loop
                }
```

```
            }
        }
    }
}

void CentroidGenerator::setIndicesOrder(Matrix angleMatrix, Matrix indicesMatrix, int
nVertices, int row)
{
    double a;
    int b;
    for (int i=0; i<nVertices; i++){
        for (int j=(i+1); j<nVertices; j++){
            if (angleMatrix.elements[row][i] > angleMatrix.elements[row][j]) {
                a = angleMatrix.elements[row][i];                       //This part
make swaps, ordering least to greatest
                b = indicesMatrix.elements[row][i];                     //for the
angles, and aligns corrsponding indices
                angleMatrix.elements[row][i] = angleMatrix.elements[row][j];
                indicesMatrix.elements[row][i] = indicesMatrix.elements[row][j];
                angleMatrix.elements[row][j] = a;
                indicesMatrix.elements[row][j] = b;
            }
        }
    }
}

void CentroidGenerator::sortIndices(Matrix sitesPosition, Matrix indicesMatrix, int
nSites, int nVertices, Matrix angleMatrix)
{
    double xyVector[2];
    for (int i=0; i<nSites; i++){
        for (int j=0; j<nVertices; j++){
            if (indicesMatrix.elements[i][j] != 0) {
                for (int k=0; k<2; k++) //xyVector[0] = xVertice-xSite  and
xyVector[1] = yVertice-ySite
                {
                    xyVector[k] =
( uniqueVertices.elements[ int(indicesMatrix.elements[i][j]-1) ][k] -
sitesPosition.elements[i][k] );
                }
                angleMatrix.elements[i][j] = atan2 (xyVector[1],xyVector[0]) * (180.00
/ PI);
            }
        }
        setIndicesOrder(angleMatrix, indicesMatrix, nVertices, i);
    }
    //indicesMatrix.printArray("Indices in Order");
    //angleMatrix.printArray("Angle Matrix");
}

void CentroidGenerator::getLocalIndex(Matrix indicesMatrix, int nVertices, int row,
vector<int>& localIndex, int& noLocVertices)
{
    for (int j=0; j<nVertices; j++)     //store indices different from zero
    {
        if (indicesMatrix.elements[row][j] != 0) localIndex.push_back
(indicesMatrix.elements[row][j]);
    }
    localIndex.push_back(localIndex.front());   //at the end, store the first element
from the vector again
    noLocVertices = int(localIndex.size());
}
```

```
void CentroidGenerator::getCentroid(int nLocalVertices, vector<int>& localIndex,
Matrix centroidMatrix, int row)
{
    double xsum = 0, ysum = 0, area = 0;

    //Get local vertices
    Matrix localVerticesPos(nLocalVertices,2);
    for (int i=0; i<nLocalVertices; i++){
        for (int j=0; j<2; j++){
            localVerticesPos.elements[i][j] =
(uniqueVertices.elements[ (localIndex[i]-1) ][j]);
        }
    }

    //Compute Area
    for (int i=0; i<(nLocalVertices-1); i++){
        area = area + localVerticesPos.elements[i][0]*localVerticesPos.elements[i+1]
[1] - localVerticesPos.elements[i+1][0]*localVerticesPos.elements[i][1];
    }
    area = area/2;

    //Compute Centroid
    for (int i=0;  i<(nLocalVertices-1); i++){
        xsum = xsum + (localVerticesPos.elements[i][0] + localVerticesPos.elements[i
+1][0])*(localVerticesPos.elements[i][0]*localVerticesPos.elements[i+1][1] -
localVerticesPos.elements[i+1][0]*localVerticesPos.elements[i][1]);
        ysum = ysum + (localVerticesPos.elements[i][1] + localVerticesPos.elements[i
+1][1])*(localVerticesPos.elements[i][0]*localVerticesPos.elements[i+1][1] -
localVerticesPos.elements[i+1][0]*localVerticesPos.elements[i][1]);
    }
    centroidMatrix.elements[row][0] = ( 1/(6*area) )*xsum;
    centroidMatrix.elements[row][1] = ( 1/(6*area) )*ysum;
}


Matrix CentroidGenerator::generateCentroid(std::vector<float> allVertices, Matrix
sitesPosition, int nSites, float minX, float maxX, float minY, float maxY, int
nCzyBdyVert)
{
    getUniqVertices(allVertices);

    Matrix indicesMatrix(nSites, uniqueVertices.rows,-1); //initialize IndicesMatrix
with value(-1). It'll store indices of vertices related to each site.
    getIndices(indicesMatrix, sitesPosition, nSites, uniqueVertices.rows);

    setVerticesOnBoundary(indicesMatrix, minX, maxX, minY, maxY, nCzyBdyVert);

    //uniqueVertices.printArray("New Unique Vertices");

    Matrix angleMatrix(nSites, uniqueVertices.rows);
    sortIndices(sitesPosition, indicesMatrix, nSites, uniqueVertices.rows,
angleMatrix);

    Matrix centroidMatrix(nSites, 2);

    int nLocalVertices = 0;
    for (int siteNumber=0; siteNumber<nSites; siteNumber++){
        vector<int> localIndex;

        //Get Local Index: Supporting function. Returns: (localIndex, noLocIndices)
        getLocalIndex(indicesMatrix, uniqueVertices.rows, siteNumber, localIndex,
nLocalVertices);

        /*cout << endl;
```

```
        cout << "DICTIONARY FOR SITE " << (siteNumber + 1) << ":" << endl;
        cout << "noLocVertices = "<< nLocalVertices << endl;
        cout << "localIndex    = ";
        for (int i=0; i<nLocalVertices; i++){
            cout << localIndex[i] << ", ";
        }
        cout << endl;
        */

        //GET CENTROID: Returns (Centroid)
        getCentroid(nLocalVertices, localIndex, centroidMatrix, siteNumber);
        /*cout << "Centroid      = ";
        cout << "(" << centroidMatrix.elements[siteNumber][0] << ", " <<
centroidMatrix.elements[siteNumber][1] << ")";
        cout << endl;*/
    }
    //cout << endl;
    centroidMatrix.printArray("Centroid");


    //print Centroid Matrix to work with Matlab
    /*cout << "cX=[";
    for (int i=0; i<centroidMatrix.rows; i++){
        if (i==centroidMatrix.rows-1) cout << centroidMatrix.elements[i][0] << "];" <<
endl;
        else cout << centroidMatrix.elements[i][0] << ", "; }
    cout << "cY=[";
    for (int i=0; i<centroidMatrix.rows; i++){
        if (i==centroidMatrix.rows-1) cout << centroidMatrix.elements[i][1] << "];" <<
endl;
        else cout << centroidMatrix.elements[i][1] << ", "; }
    */

    return centroidMatrix;
}
int main(int argc, const char * argv[]) {
    //variables provided by publisher
    const int count = 5;    //number of sites(robots)
    float xValues[count] = {4, 5, 19, 36, 51};    //X position of sites(robots)
    float yValues[count] = {13, 49, 88, 76, 27};    //Y position of sites(robots)

    const int nCzyBdyVert = 8;
    float boundaryPos[nCzyBdyVert][2] = {{2, -15}, {35, -23}, {72, -23}, {112, -3},
{109, 98}, {100, 120}, {-5, 120}, {-43, 36}};
    //float xBoundary[nCzyBdyVert] = {2, 35, 72, 112, 109, 100, -5, -43};
    //float yBoundary[nCzyBdyVert] = {-15, -23, -23, -3, 98, 120, 120, 36};

    //define the minimum and maximum X and Y values for the Fortune's Algorithm
    float minX, maxX, minY, maxY;
    for (int i=0; i<nCzyBdyVert; i++) {
        if (i==0) {
            minX = maxX = boundaryPos[i][0];
            minY = maxY = boundaryPos[i][1];
        }
        else {
            if (boundaryPos[i][0]<minX) minX = boundaryPos[i][0];
            else if(boundaryPos[i][0]>maxX) maxX = boundaryPos[i][0];
            if (boundaryPos[i][1]<minY) minY = boundaryPos[i][1];
            else if(boundaryPos[i][1]>maxY) maxY = boundaryPos[i][1];
        }
    }
    //cout << "minX: " << minX << ". maxX: " << maxX << ". minY: " << minY << ". maxY:
" << maxY << endl;
    //float minX = -43, maxX = 112;
```

```
    //float minY = -23, maxY = 120;

    //Store the position of the sites in a Matrix
    int nSites = Matrix_Size(xValues);
    Matrix sitesPos(nSites,2);
    for(int i=0; i<Matrix_Size(xValues);i++){
        sitesPos.setElement(i, 0, xValues[i]);      //sitePos.elements[i][0] =
xValues[i];
        sitesPos.setElement(i, 1, yValues[i]);
    }
    //sitesPos.printArray("Sites");

    int iteration=1;
    while (iteration<=100)
    {
        cout << endl << "Iteration " << iteration;
        CentroidGenerator cg;
        VoronoiDiagramGenerator vdg;
        vdg.generateVoronoi(xValues, yValues, count, minX, maxX, minY, maxY, 3);

        vdg.resetIterator();

        float x1,y1,x2,y2;
        int a=1;
        printf("\n-------------------------------\n");
        while(vdg.getNext(x1,y1,x2,y2))
        {
            //printf("GOT Line (%.4f,%.4f)->(%.4f,%.4f)\n", x1,y1,x2,y2);
            printf("v%dx = [%.4f,%.4f];\n", a, x1,x2);
            printf("v%dy = [%.4f,%.4f];\n", a, y1,y2);
            a++;
            if (x1!=x2 || y1!=y2)  //if condition necessary due to some unknown
problem (Fortune's Algorithm generating vertices that shouldn't exist)
            {
                cg.posVertVector.push_back(x1); cg.posVertVector.push_back(y1);
                cg.posVertVector.push_back(x2); cg.posVertVector.push_back(y2);
            }                       //even though it seems to not affect the
centroids' position
        }
        cout << endl;

        //After store position of all vertices, store the position of the edges of the
plane (polygon boundary)
        for (int i=0; i<nCzyBdyVert; i++) {
            cg.posVertVector.push_back(boundaryPos[i][0]);
            cg.posVertVector.push_back(boundaryPos[i][1]);
        }


        //Return the position of the centroids
        sitesPos = cg.generateCentroid(cg.posVertVector, sitesPos, nSites, minX, maxX,
minY, maxY, nCzyBdyVert);
        //sitesPos.printArray("new");

        //Split the Centroid Matrix in X and Y vectors in order to pass in to the
Fortune's Algorithm
        for (int i=0; i<sitesPos.rows; i++) {
            xValues[i]=sitesPos.elements[i][0];
            yValues[i]=sitesPos.elements[i][1];
        }
        iteration++;
    }
    return 0;
}
```

CentroidGenerator_CzyBdy.h

```
#ifndef __Centroid_Generator__
#define __Centroid_Generator__

#include <iostream>
#include <stdio.h>
#include <vector>
#include <math.h>

#define Matrix_Size(x) (sizeof(x) / sizeof(x[0]))
#define PI 3.14159265

class Matrix {
public:
    Matrix(int r, int c, float data=0);
    ~Matrix();
    void setMatrix(int r, int c, float data=0);
    int setCols(int c);//to modify the columns
    int setRows(int r);//to modify the rows
    void printArray(std::string name="Matrix");
    void setElement(int r, int c, float data);

    int rows, cols;
    float **elements;
};


class CentroidGenerator
{
public:
    //CentroidGenerator();
    ~CentroidGenerator();
    std::vector<float> posVertVector;

    float truncate(float num, int precision);
    float distance(float siteX, float siteY, float vertX, float vertY);
    void setIndicesOrder(Matrix angleMatrix, Matrix indicesMatrix, int nVertices, int
row);

    void getUniqVertices(std::vector<float> allVerticesVector);
    void getIndices(Matrix indicesMatrix, Matrix Sites, int nSites, int nVertices);

    void findVerticesOnCrazyBdy(float x1, float y1, float x0, float y0, int row, int
nCzyBdyVert);
    void setVerticesOnBoundary(Matrix indicesMatrix, float minX, float maxX, float
minY, float maxY, int nCzyBdyVert);

    void sortIndices(Matrix sitesPosition, Matrix indicesMatrix, int nSites, int
nVertices, Matrix angleMatrix);

    void getLocalIndex(Matrix indicesMatrix, int nVertices, int row, std::vector<int>&
localIndex, int& noLocVertices);
    void getCentroid(int nLocalVertices, std::vector<int>& localIndex, Matrix
centroidMatrix, int row);

    Matrix generateCentroid(std::vector<float> allVertices, Matrix sitesPosition, int
nSites, float minX, float maxX, float minY, float maxY, int nCzyBdyVert);
};

#endif
```