**Project Report**

**Introduction:**

The goal of this practical project was to implement a program to produce the Delaunay triangulation and Voronoi diagram of a given set of input points.

Voronoi diagrams are a concept with important applications in various fields including statistics, epidemiology, ecology, or machine learning. They are so useful because they allow spatially distributed information to be processed and analyze. Delaunay triangulations can be used for the construction of Voronoi diagrams because the two graphs are duals of each other. By themselves, Delaunay triangulations are useful for the in 3D modeling because they avoid the construction of narrow triangles.

The implementation described in this report and presented in the program listings is capable of providing graphical output of both Delaunay triangulation and Voronoi diagram based on user’s preference, while the number of points to analyze, and the points themselves can be supplied by the user.

**Background:**

A Voronoi diagram of *n* input points divides the plane into *n* regions. All points inside a given region are closer to a particular input point than any other one. Voronoi diagrams were developed by Ukraine-born mathematician Georgy Voronoy at the turn of the nineteenth century. They have important applications in several areas of human endeavor. In statistics, they allow the interpolation of geographically distributed data points. They are also used in epidemiology, to locate a source of infection; in ecology, to describe growth patterns of forests and tree canopies; and in machine learning, for determining the 1-NN classifications of sample data in parameter space. For their visual appearance, Voronoi diagrams have also been used as a motif in architecture and art.

One of Voronoy’s students, Boris Delaunay, developed the dual graph of the Voronoi diagram. It is the Delaunay triangulation; a triangulation of a set of points, in which two sets are connected only if they share a common boundary in a Voronoi diagram. This is the angle-maximizing triangulation of a set of points. As such, it tends to avoid narrow triangles, which makes the concept useful in the modeling of terrain based on a set of input points. The triangulation is also used to refine the data provided by meshing algorithms, so that the data can be presented well on-screen.

Nevertheless, the Delaunay criterion is independent of concept of Voronoi diagram altogether. Rather, it is defined in terms of the circumcircles of the triangular faces that are constructed by the triangulation – the circumcircle of any face must be empty for the triangulation to pass the Delaunay criterion. If this is not the case, and there is only one extra point inside the circumcircle, a flip operation can be performed between the two corresponding triangular faces. The shared boundary is cut and replaced by the roughly perpendicular connection between the two vertices across the triangle. The two triangular faces now pass the Delaunay criterion.

Furtheromore, the two concepts are related to the notion of the convex hull in the third dimension. If set of input points is projected onto a parabolic surface and connected according to the convex hull rules, the resulting triangulation is the Delaunay triangulation. This observation is significant, because it places the lower bound of any algorithm to calculate the Delaunay triangulation of *n* points at Ω(*n*log*n*).

**Implementation:**

The program makes use of several objects:

* Lists are used to store pointers to Points, HalfEdges and Triangles. It contains a start Node and has an integer size. Data can be added to the List through the insert() method. The List can also remove Nodes, either directly or indirectly – by first finding the Node based on the data provided and then removing that Node.
* Node is used by the List to hold the List data as well as the pointers to previous and next nodes.
* A Point stores double x and y coordinate, as well as an integer counter and a double random\_id. Furthermore, it stores a pointer to the triangle it is in, as well as to its Node in Delaunay’s List of points. It also keeps a list of pointers to HalfEdges that originate at the point. Furthermore, it contains a method to determine its distance from another point.
* HalfEdge keeps a pointer to its Point source and target, as well as its twin and next HalfEdge. HalfEdge is one of the crucial data structures of this program, because they contain all the information necessary to construct the Delaunay triangulation. In this program, a layer of abstraction has been added through the inclusion of the Triangle data structure (explained below), but that is not necessary. HalfEdges are of crucial importance during the necessary flip() and split() operations (explained below).
* A Triangle stores three HalfEdges that together form one of the triangular faces of the triangulation. It also holds the input points that still need to be inserted into the triangulation.
* Delaunay is the overarching object for the input reading operations, insertion of the points into the triangulation, and the writing of the output into a file.

The algorithm works in the following way (note: whenever “edge” is referenced it is taken to mean “the HalfEdge and its twin”):

* First, create 4 helper points in the four corners of the input area (the rectangle from (0,0) to (100,100) inclusive) and make two helper triangles.
* Then, read the input file and parse the x and y coordinate data. Make new points based on these data and assign a random\_id to each point.
* Next, sort the points based on their random\_id values. This is to ensure that points are inserted into the triangulation in a random order and thus the expected number of flips is 1, which keeps the complexity at O(*n*log*n*).
* Add the – now randomly sorted – input points into the corresponding triangles of the triangulation. Initially, these triangles are the two triangles made of helper points.
* Select input points one by one (a list of the unassigned input points is kept by the Delaunay object) until there are no unassigned input points left.
  + The point is inserted by the split operation. Connect the point to the three vertices of the triangle inside which it lies by HalfEdges (including the twin HalfEdges in the opposite direction). Make three new triangles. Reassign the input points contained in the original triangle, so that they are included in the correct new triangle. After this, delete the original triangle.
  + It is possible that the creation of new triangles has left the triangulation is a non-Delaunay-condition-conforming state. To remedy that, it may be necessary to flip one or more of the edges of the original triangle (now contained in the new triangles). Obtain the triangle across the edge (by referring to the HalfEdge’s twin’s of\_triangle attribute) and obtain reference to the point that does not lie on the edge in question. Connect that point to the point on the other side of the edge in question (in this case, this point is the point we are inserting to the triangulation). Insert the resulting HalfEdges into Delaunay’s list of HalfEdges and remove the offending edge.
  + It is possible that the reassignment of edges necessitate flips. That is why we flip recursively, until no flip is needed.
  + After all the flips are performed, the triangulation conforms to the Delaunay criterion once again.
* After all the points are inserted, create output based on the command line arguments passed by the user (specifying whether the Delaunay triangulation should be output, and whether the Voronoi diagram should be output). The output is made to be run by the gnuplot utility. (The steps to take are detailed in the README file.)

**Evaluation:**

The program expects any number of inputs in the form of a <number>\t<number> in the range of 0 to 100 (inclusive), given in an input file. Each of the input points records is separated by a newline (\n). The address of the input file may be specified by the user.

If any one of two values of an input point lies outside the expected range, the program prints a corresponding warning in the console output.

If the information is not provided in the expected format, the program is not able to parse it and informs the user of that fact in the console output. If the data provided matches the formatting expected (including the \t’s and \n’s), the program tries to parse the information, and returns 0.0 for the unparseable values.

If the points given are not in the general position, the program may fail, since input points lying exactly on the half-edges will not be assigned to either side triangle during flips or splits (this measure is for consistency within the program). The program may not fail in every such case (especially if the points lying on one line are far apart), but if it does, the user is informed of the error in the console output.

The program is able to produce a Delaunay triangulation for 1,000 points in a short time. As a limitation, however, the triangulation of 10,000 points already takes several minutes.

**Program Listing:**

Please also refer to the files in the attached .zip file (zbynek.stara-DS Final Project.zip). Among these is the randinput.cpp random point generator, bash scripts to allow easy building and running of the C++ source files (on Mac OS X). The README.md contains instructions on how to use the files as well as the expected data format of the input file. Any number of new random inputs can be generated on demand, which is why only one sample input file is provided.

Reproduced below is the code of the main delaunay.cpp file:

#include <iostream>

#include <fstream>

#include <string>

#include <cstring>

#include <cmath>

#include <sstream>

#include <cstdlib>

#include <time.h>

#include <new>

using namespace std;

template <typename T>

class Node {

public:

T data;

Node<T>\* next;

Node<T>\* previous;

Node(T data) {

this->data = data;

this->next = NULL;

this->previous = NULL;

}

};

template <typename T>

class List {

public:

int size;

Node<T>\* start;

List() {

start = NULL;

size = 0;

}

Node<T>\* insert(T data) {

Node<T>\* new\_node = new Node<T>(data);

// adding new node to the front of the list

new\_node->next = start;

new\_node->previous = NULL;

if (start != NULL) {

start->previous = new\_node;

}

start = new\_node;

size += 1;

return new\_node;

}

void remove(Node<T>\* node) {

if (node != NULL) {

// removes the node provided

if (node == start) {

start = start->next;

}

if (node->next != NULL) {

node->next->previous = node->previous;

}

if (node->previous != NULL) {

node->previous->next = node->next;

}

delete node;

size -= 1;

}

else {

// we are deleting nothing

return;

}

}

void remove\_data(T data) {

// first search for the node and then remove it

remove(find(data));

}

Node<T>\* find(T data) {

if (size == 0) {

// it cannot be there

return NULL;

}

else {

Node<T>\* current = start;

// go through the nodes and see the contents

while (current != NULL) {

if (current->data == data) {

return current;

} else {

current = current->next;

}

}

// if it is not encountered

return NULL;

}

}

};

class HalfEdge;

class Triangle;

class Point { // stores points\_node

public:

double x;

double y;

int counter;

double random\_id; // for sorting of the input points to ensure randomness

Triangle\* in\_triangle; // in which triangle it originally is

Node<Point\*>\* points\_node; // the node of this in points list

List<HalfEdge\*>\* half\_edges; // half-edges containing this point

Point(double x, double y, int counter, double random\_id) {

this->x = x;

this->y = y;

this->counter = counter;

this->random\_id = random\_id;

in\_triangle = NULL;

points\_node = NULL;

half\_edges = new List<HalfEdge\*>();

}

~Point() {

delete half\_edges;

}

double distance\_from(Point\* b) {

// euclidean distance

double ax = this->x;

double bx = b->x;

double ay = this->y;

double by = b->y;

double adx = abs(ax - bx);

double ady = abs(ay - by);

return sqrt(pow(adx, 2.0) + pow(ady, 2.0));

}

string to\_string() {

stringstream s;

s << "(" << x << ", " << y << ")";

return s.str();

}

// two points are equal if they have the same x and y

bool operator== (const Point\* rhs) const {

return ((this->x==rhs->x) && (this->y==rhs->y));

}

bool operator!= (const Point\* rhs) const {

return !(this==rhs);

}

};

class HalfEdge { // stores half\_edges\_node

public:

Point\* source;

Point\* target;

HalfEdge\* twin;

HalfEdge\* next;

bool helper; // useful for delaunay output

Triangle\* of\_triangle;

Node<HalfEdge\*>\* half\_edges\_node; // what node in half-edges list this is

Point\* midpoint;

HalfEdge(Point\* source, Point\* target) {

this->source = source;

this->target = target;

((this->source)->half\_edges)->insert(this);

twin = NULL;

next = NULL;

// is this a helper?

helper = false;

if (source->counter < 0 || target->counter < 0) {

helper = true;

}

of\_triangle = NULL;

half\_edges\_node = NULL;

double mx = (target->x+source->x)/2.0;

double my = (target->y+source->y)/2.0;

midpoint = new Point(mx,my,-1234,-1234);

}

~HalfEdge() {

((this->source)->half\_edges)->remove\_data(this);

// fails quietly if not there

delete midpoint;

}

// helper function

double area(Point\* p) {

// positive = true if anticlockwise direction

Point\* a = this->source;

Point\* b = this->target;

Point\* c = p;

double xa = a->x;

double ya = a->y;

double xb = b->x;

double yb = b->y;

double xc = c->x;

double yc = c->y;

double sum = (0.5)\*((xa\*(yb-yc))+(xb\*(yc-ya))+(xc\*(ya-yb)));

return sum;

// if it is zero, it will not get included

// if a point is on the line, it is not in any triangle

// therefore, input\_points have to be given general position

}

string to\_string() {

stringstream s;

s << "(" << source->counter << ", " << target->counter << ")";

return s.str();

}

// two half-edges are equal if they have the same source and target

bool operator== (const HalfEdge\* rhs) const {

return ((this->source==rhs->source) && (this->target==rhs->target));

}

bool operator!= (const HalfEdge\* rhs) const {

return !(this==rhs);

}

};

class Triangle { // stores triangles\_node

public:

HalfEdge\* ab; // primary half\_edge

HalfEdge\* bc;

HalfEdge\* ca;

bool helper; // useful for voronoi

Point\* center; // used if voronoi option employed

double radius;

Node<Triangle\*>\* triangles\_node; // the node of this in triangles list

List<Point\*>\* points; // list of unassigned points in the triangle

Triangle(HalfEdge\* ab, HalfEdge\* bc, HalfEdge\* ca) {

// is this a helper?

// if at least one half-edge is a helper, this is a helper

helper = (ab->helper || bc->helper || ca->helper);

// connect half-edges together

ab->next = bc;

bc->next = ca;

ca->next = ab;

// add the half-edges to the triangle

this->ab = ab;

ab->of\_triangle = this;

this->bc = bc;

bc->of\_triangle = this;

this->ca = ca;

ca->of\_triangle = this;

// calculate center

Point\* a = ab->source;

Point\* b = bc->source;

Point\* c = ca->source;

double ax = a->x;

double ay = a->y;

double bx = b->x;

double by = b->y;

double cx = c->x;

double cy = c->y;

double d = 2\*(ax\*(by-cy) + bx\*(cy-ay) + cx\*(ay-by));

double ux = ((ax\*ax + ay\*ay) \* (by-cy)

+ (bx\*bx + by\*by) \* (cy-ay)

+ (cx\*cx + cy\*cy) \* (ay-by))

/ d;

double uy = ((ax\*ax + ay\*ay) \* (cx-bx)

+ (bx\*bx + by\*by) \* (ax-cx)

+ (cx\*cx + cy\*cy) \* (bx-ax))

/ d;

center = new Point(ux,uy,-999,-999);

// radius is the distance between a point and the center

radius = a->distance\_from(center);

triangles\_node = NULL;

points = new List<Point\*>();

}

~Triangle() {

delete center;

delete points;

}

string to\_string() {

stringstream s;

s << "(" << ab->source->counter << ", "

<< bc->source->counter << ", "

<< ca->source->counter << ")";

return s.str();

}

};

class Delaunay { // includes delaunay and voronoi

private:

List<Point\*>\* input\_points;

List<Point\*>\* random\_input\_points; // shuffled input\_pts to ensure O(nlogn)

List<Point\*>\* points;

List<HalfEdge\*>\* half\_edges;

List<Triangle\*>\* triangles;

string input\_address;

string output\_address;

bool delaunay;

bool voronoi;

public:

Delaunay(string input\_address, string output\_address,

bool delaunay, bool voronoi) {

// seed the random number generator

srand(time(NULL));

input\_points = new List<Point\*>();

random\_input\_points = new List<Point\*>();

points = new List<Point\*>();

half\_edges = new List<HalfEdge\*>();

triangles = new List<Triangle\*>();

this->input\_address = input\_address;

this->output\_address = output\_address;

this->delaunay = delaunay;

this->voronoi = voronoi;

// make initial helper points

Point\* a = new Point(0,0,-1,-1);

Point\* b = new Point(100,0,-2,-1);

Point\* c = new Point(0,100,-3,-1);

Point\* d = new Point(100,100,-4,-1);

// include these helper points in the points list

a->points\_node = points->insert(a);

b->points\_node = points->insert(b);

c->points\_node = points->insert(c);

d->points\_node = points->insert(d);

// add helper half edges to the half edges list

HalfEdge\* ab = new HalfEdge(a,b);

ab->half\_edges\_node = half\_edges->insert(ab);

HalfEdge\* bc = new HalfEdge(b,c);

bc->half\_edges\_node = half\_edges->insert(bc);

HalfEdge\* ca = new HalfEdge(c,a);

ca->half\_edges\_node = half\_edges->insert(ca);

HalfEdge\* dc = new HalfEdge(d,c);

dc->half\_edges\_node = half\_edges->insert(dc);

HalfEdge\* cb = new HalfEdge(c,b);

cb->half\_edges\_node = half\_edges->insert(cb);

HalfEdge\* bd = new HalfEdge(b,d);

bd->half\_edges\_node = half\_edges->insert(bd);

// add twins

bc->twin = cb;

cb->twin = bc;

// creating the helper triangles from the new half edges

Triangle\* triangle1 = new Triangle(ab,bc,ca);

triangle1->triangles\_node = triangles->insert(triangle1);

Triangle\* triangle2 = new Triangle(dc,cb,bd);

triangle2->triangles\_node = triangles->insert(triangle2);

// input the points from file address

read\_file(input\_address);

// shuffle input points

shuffle\_input();

}

~Delaunay() {

// delete lists

// also delete all contents of the lists

// delete input points

Node<Point\*>\* current\_p\_node = input\_points->start;

while (input\_points->size != 0) {

Node<Point\*>\* old\_current\_p\_node = current\_p\_node;

current\_p\_node = current\_p\_node->next;

input\_points->remove(old\_current\_p\_node);

}

delete input\_points;

// delete random input points

current\_p\_node = random\_input\_points->start;

while (random\_input\_points->size != 0) {

Node<Point\*>\* old\_current\_p\_node = current\_p\_node;

current\_p\_node = current\_p\_node->next;

random\_input\_points->remove(old\_current\_p\_node);

}

delete random\_input\_points;

// delete points

current\_p\_node = points->start;

while (points->size != 0) {

Node<Point\*>\* old\_current\_p\_node = current\_p\_node;

current\_p\_node = current\_p\_node->next;

points->remove(old\_current\_p\_node);

}

delete points;

// delete half edges

Node<HalfEdge\*>\* current\_e\_node = half\_edges->start;

while (half\_edges->size != 0) {

Node<HalfEdge\*>\* old\_current\_e\_node = current\_e\_node;

current\_e\_node = current\_e\_node->next;

half\_edges->remove(old\_current\_e\_node);

}

delete half\_edges;

// delete triangles

Node<Triangle\*>\* current\_t\_node = triangles->start;

while (triangles->size != 0) {

Node<Triangle\*>\* old\_current\_t\_node = current\_t\_node;

current\_t\_node = current\_t\_node->next;

(triangles)->remove(old\_current\_t\_node);

}

delete triangles;

}

void read\_file(string file\_address) {

// read the input file and put points into list

// the input file has the format of x\_coordinate\ty\_coordinate

string line;

ifstream input\_file(file\_address.c\_str(), ios::in);

if (input\_file.is\_open()) {

int counter = 0;

while (getline(input\_file, line)) {

// split the string to strings with x and y coordinates

int pos = line.find("\t");

string x\_str = line.substr(0, pos);

string y\_str = line.substr(pos);

// read the numbers

// we assume valid inputs - if invalid, returns 0.0

double x = atof(x\_str.c\_str());

double y = atof(y\_str.c\_str());

// make a random double

double random\_id = (double) rand() / RAND\_MAX;

// create corresponding point and insert to input\_points

if (x <= 100 && x >= 0 && y <= 100 && y >= 0) {

Point\* p = new Point(x, y, counter, random\_id);

input\_points->insert(p);

// put the input point in correct triangle

Node<Triangle\*>\* current = triangles->start;

while (current != NULL) {

Triangle\* t = current->data;

if (is\_in(t, p)) {

(t->points)->insert(p);

p->in\_triangle = t;

break;

}

current = current->next;

}

}

else {

// if the point is out of range

cout << "Warning: Point with counter " << counter

<< " was ignored!\n";

}

counter += 1;

}

input\_file.close();

}

else {

cout << "Error: Could not open input file!\n";

}

}

void shuffle\_input() {

// sorts according to points' random\_id's

// since the id's are random, it makes the input order random

// uses quicksort

if (input\_points->size > 0) {

// make a point\* array of input\_points

Point\*\* input = new (nothrow) Point\* [input\_points->size];

Node<Point\*>\* current = input\_points->start;

int counter = 0;

while (current != NULL) {

// add the input\_points to the array one by one

input[counter] = current->data;

current = current->next;

counter += 1;

}

// perform quicksort on the input array

input\_quicksort(input, 0, input\_points->size-1);

for (int i = 0; i < input\_points->size; i++) {

// insert input\_points to random\_input\_points one by one

random\_input\_points->insert(input[i]);

}

// delete the input array

delete[] input;

}

}

// the recursive sort operation

void input\_quicksort(Point\*\* input, int l, int r) {

// the array is changed in-place

int orig\_l = l;

int orig\_r = r;

int size = r - l;

double pivot = input[orig\_l]->random\_id;

// swap things around while pointers are at correct relative positions

while (l <= r && l < size) {

bool l\_stayed = true;

bool r\_stayed = true;

if (input[l]->random\_id <= pivot) {

l += 1;

l\_stayed = false;

}

if (input[r]->random\_id > pivot) {

r -= 1;

r\_stayed = false;

}

// if both pointers cannot move - we can swap

if (l\_stayed && r\_stayed) {

Point\* swap = input[l];

input[l] = input[r];

input[r] = swap;

}

}

// finally, we swap the right pointer's current position with first

Point\* swap = input[r];

input[r] = input[orig\_l];

input[orig\_l] = swap;

// do quicksort for the two resulting paths

if (r != orig\_l) {

input\_quicksort(input, orig\_l, r-1);

}

if (l < size) {

input\_quicksort(input, l, orig\_r);

}

}

bool is\_in(Triangle\* t, Point\* p) {

// uses the sign of areas

// triangle is made of each half edge and p

// if the sign of all of these triangles is the same = inside

// if at least one of them is different = outside

HalfEdge\* ab = t->ab;

HalfEdge\* bc = t->bc;

HalfEdge\* ca = t->ca;

return (((ab->area(p) > 0) == (bc->area(p) > 0)) &&

((bc->area(p) > 0) == (ca->area(p) > 0)));

}

bool circumcircle\_empty(Triangle\* t) {

bool val = true;

Node<Point\*>\* current\_p\_node = points->start;

// go through all points

// could be more localized

while (current\_p\_node != NULL) {

Point\* p = current\_p\_node->data;

if (p != t->ab->source &&

p != t->bc->source &&

p != t->ca->source) {

// only if p is not one of the points of the triangle

if (p->distance\_from(t->center) < t->radius) {

// when it's on the circumcircle, it doesn't count

val = false;

// one of the points is in = circumcircle not empty

break;

}

}

current\_p\_node = current\_p\_node->next;

}

return val;

}

void split\_triangle(Triangle\* t, Point\* p) {

// split that triangle into three new triangles

// therefore, make three triangles inside and connect

// edges of the old triangle

HalfEdge\* old\_e1 = t->ab;

HalfEdge\* old\_e2 = t->bc;

HalfEdge\* old\_e3 = t->ca;

// make new edges and insert them to half edges list

HalfEdge\* new\_e1\_1 = new HalfEdge(p, old\_e1->source);

HalfEdge\* new\_e1\_2 = new HalfEdge(old\_e1->source, p);

new\_e1\_1->twin = new\_e1\_2;

new\_e1\_2->twin = new\_e1\_1;

new\_e1\_1->half\_edges\_node = half\_edges->insert(new\_e1\_1);

new\_e1\_2->half\_edges\_node = half\_edges->insert(new\_e1\_2);

HalfEdge\* new\_e2\_1 = new HalfEdge(p, old\_e2->source);

HalfEdge\* new\_e2\_2 = new HalfEdge(old\_e2->source, p);

new\_e2\_1->twin = new\_e2\_2;

new\_e2\_2->twin = new\_e2\_1;

new\_e2\_1->half\_edges\_node = half\_edges->insert(new\_e2\_1);

new\_e2\_2->half\_edges\_node = half\_edges->insert(new\_e2\_2);

HalfEdge\* new\_e3\_1 = new HalfEdge(p, old\_e3->source);

HalfEdge\* new\_e3\_2 = new HalfEdge(old\_e3->source, p);

new\_e3\_1->twin = new\_e3\_2;

new\_e3\_2->twin = new\_e3\_1;

new\_e3\_1->half\_edges\_node = half\_edges->insert(new\_e3\_1);

new\_e3\_2->half\_edges\_node = half\_edges->insert(new\_e3\_2);

// make new triangles using the old and new edges

Triangle\* new\_t\_1 = new Triangle(old\_e1,new\_e2\_2,new\_e1\_1);

new\_t\_1->triangles\_node = triangles->insert(new\_t\_1);

Triangle\* new\_t\_2 = new Triangle(old\_e2,new\_e3\_2,new\_e2\_1);

new\_t\_2->triangles\_node = triangles->insert(new\_t\_2);

Triangle\* new\_t\_3 = new Triangle(old\_e3,new\_e1\_2,new\_e3\_1);

new\_t\_3->triangles\_node = triangles->insert(new\_t\_3);

// reallocate the remaining input\_points in original triangle

// so that they are still in a corresponding triangle

// also, remove p from old triangle's input\_points

// loop through the triangle's points

Node<Point\*>\* current\_t\_p\_node = t->points->start;

while ((t->points)->size != 0) {

Point\* t\_p = current\_t\_p\_node->data;

if (t\_p != p) { // one of the points returned will be p itself

if (is\_in(new\_t\_1, t\_p)) {

(new\_t\_1->points)->insert(t\_p);

t\_p->in\_triangle = new\_t\_1;

}

else if (is\_in(new\_t\_2, t\_p)) {

(new\_t\_2->points)->insert(t\_p);

t\_p->in\_triangle = new\_t\_2;

}

else if (is\_in(new\_t\_3, t\_p)) {

(new\_t\_3->points)->insert(t\_p);

t\_p->in\_triangle = new\_t\_3;

}

else {

cout << "Error: Wrong point position.\n";

// Can happen if points are not in general position

}

}

// remove the p from the triangle

Node<Point\*>\* old\_current\_t\_p\_node = current\_t\_p\_node;

current\_t\_p\_node = current\_t\_p\_node->next;

(t->points)->remove(old\_current\_t\_p\_node);

}

// for each half-edge of the triangle, check for flip

// if there are flips, check the adjacent half-edges too = recurse

// (inside the flip function, reallocate the input\_points)

flip(old\_e1);

flip(old\_e2);

flip(old\_e3);

// finally, delete the original big triangle

triangles->remove(t->triangles\_node);

}

void flip(HalfEdge\* ab) {

// switching to the most advantageous edge - flipping if necessary

// definitions that are for sure

Point\* c = (ab->next)->target;

HalfEdge\* bc = ab->next;

HalfEdge\* ca = bc->next;

Triangle\* old\_1 = ab->of\_triangle;

if (ab->twin != NULL && !circumcircle\_empty(old\_1)) {

// it's enough to check one triangle

// if one triangle fails, they both do

// definitions that only apply if a twin exists

Point\* d = ((ab->twin)->next)->target;

HalfEdge\* ba = ab->twin;

HalfEdge\* ad = ba->next;

HalfEdge\* db = ad->next;

Triangle\* old\_2 = ba->of\_triangle;

// make new edges across

HalfEdge\* cd = new HalfEdge(c, d);

HalfEdge\* dc = new HalfEdge(d, c);

// make them twins

cd->twin = dc;

dc->twin = cd;

// insert them into half edges list

cd->half\_edges\_node = half\_edges->insert(cd);

dc->half\_edges\_node = half\_edges->insert(dc);

// make two new triangles

Triangle\* t\_1 = new Triangle(db,bc,cd);

t\_1->triangles\_node = triangles->insert(t\_1);

Triangle\* t\_2 = new Triangle(ad,dc,ca);

t\_2->triangles\_node = triangles->insert(t\_2);

// transfer input\_points to these triangles

// from the first old triangle

Node<Point\*>\* current\_o\_1\_p\_node = (old\_1->points)->start;

while (old\_1->points->size != 0) {

Point\* o\_1\_p = current\_o\_1\_p\_node->data;

if (is\_in(t\_1, o\_1\_p)) {

// the old point is in the first new trianagle

(t\_1->points)->insert(o\_1\_p);

o\_1\_p->in\_triangle = t\_1;

}

else if (is\_in(t\_2, o\_1\_p)) {

// the old point is in the second new trianagle

(t\_2->points)->insert(o\_1\_p);

o\_1\_p->in\_triangle = t\_2;

}

else {

cout << "Error: Wrong point position.\n";

// Can happen if points are not in general position

}

// remove the point from the triangle

Node<Point\*>\* old\_current\_o\_1\_p\_node = current\_o\_1\_p\_node;

current\_o\_1\_p\_node = current\_o\_1\_p\_node->next;

(old\_1->points)->remove(old\_current\_o\_1\_p\_node);

}

// from the second old triangle

Node<Point\*>\* current\_o\_2\_p\_node = (old\_2->points)->start;

while (old\_2->points->size != 0) {

Point\* o\_2\_p = current\_o\_2\_p\_node->data;

if (is\_in(t\_1, o\_2\_p)) {

// the old point is in the first new trianagle

(t\_1->points)->insert(o\_2\_p);

o\_2\_p->in\_triangle = t\_1;

}

else if (is\_in(t\_2, o\_2\_p)) {

// the old point is in the second new trianagle

(t\_2->points)->insert(o\_2\_p);

o\_2\_p->in\_triangle = t\_2;

}

else {

cout << "Error: Wrong point position.\n";

// Can happen if points are not in general position

}

// remove the point from the triangle

Node<Point\*>\* old\_current\_o\_2\_p\_node = current\_o\_2\_p\_node;

current\_o\_2\_p\_node = current\_o\_2\_p\_node->next;

(old\_2->points)->remove(old\_current\_o\_2\_p\_node);

}

// remove old triangles

triangles->remove(old\_1->triangles\_node);

triangles->remove(old\_2->triangles\_node);

// remove old half edges

Point\* a = ab->half\_edges\_node->data->source;

Point\* b = ba->half\_edges\_node->data->source;

a->half\_edges->remove\_data(ab->half\_edges\_node->data);

half\_edges->remove(ab->half\_edges\_node);

b->half\_edges->remove\_data(ba->half\_edges\_node->data);

half\_edges->remove(ba->half\_edges\_node);

// recurse:

// the ab was set to be the so far unaffected edge

// not one of the new edges in the split triangle

// not one of the new edges made during flipping

// but the other one

// that edge needs to be checked

flip(t\_1->ab);

flip(t\_2->ab);

}

}

void insert\_points() {

// taking points one by one from the random input points

// and adding them to points

// has to re-do the delaunay connections

Node<Point\*>\* current\_p\_node = random\_input\_points->start;

while (random\_input\_points->size != 0) {

// inserting the point into points

Point\* p = current\_p\_node->data;

p->points\_node = points->insert(p);

// what triangle this input\_point is in

Triangle\* t = p->in\_triangle;

// split that triangle into three new triangles

split\_triangle(t,p);

// add another point

Node<Point\*>\* old\_current\_p\_node = current\_p\_node;

current\_p\_node = current\_p\_node->next;

// delete the input point

random\_input\_points->remove(old\_current\_p\_node);

}

}

void write\_output() {

// writes the output for gnuplot to read

ofstream output\_file(output\_address.c\_str(), ios::out);

if (output\_file.is\_open()) {

output\_file << "#!/usr/local/bin/gnuplot\n";

output\_file << "reset\n";

output\_file << "set yrange [0:100]\n";

output\_file << "set xrange [0:100]\n";

output\_file << "unset colorbox\n";

// delaunay connections:

output\_file << "set style arrow 1 nohead lc rgb \'black\'\n";

// voronoi boundaries:

output\_file << "set style arrow 2 nohead lc rgb \'red\'\n";

// go through all points

Node<Point\*>\* current\_p\_node = points->start;

while (current\_p\_node != NULL) {

// go through all of the point's half edges

Point\* p = current\_p\_node->data;

Node<HalfEdge\*>\* current\_e\_node = p->half\_edges->start;

while (current\_e\_node != NULL) {

HalfEdge\* e = current\_e\_node->data;

if (e->target->counter > p->counter || e->twin == NULL) {

// only does draw operations for one of the twins

// draw delaunay:

if (delaunay) {

// draw delaunay connections

output\_file << "set arrow from "

<< e->source->x << ","

<< e->source->y << " to "

<< e->target->x << ","

<< e->target->y

<< " as 1\n";

}

else {

// only draw delaunay points:

// for source point of half edges

output\_file << "set label "

<< (e->source->counter + 4 + 1)

<< " \"\" at "

<< e->source->x << "," << e->source->y

<< " point pointtype 5 "

<< "lc rgb \'black\'\n";

// for target point of half edge

output\_file << "set label "

<< (e->target->counter + 4 + 1)

<< " \"\" at "

<< e->target->x << "," << e->target->y

<< " point pointtype 5 "

<< "lc rgb \'black\'\n";

}

// draw voronoi:

if (voronoi && (e->twin != NULL)) {

// draw voronoi boundary between triangle centers

output\_file << "set arrow from "

<< e->of\_triangle->center->x

<< ","

<< e->of\_triangle->center->y

<< " to "

<< e->twin->of\_triangle->center->x

<< ","

<< e->twin->of\_triangle->center->y

<< " as 2\n";

}

else if (voronoi && (e->twin == NULL)) {

// draw voronoi boundary between center & midpoint

// making virtual point to which boundary will aim

double mx\_offset = 0;

double my\_offset = 0;

// determining to which direction to offset

if (e->midpoint->x == 0) {

mx\_offset -= 1000;

}

else if (e->midpoint->x == 100) {

mx\_offset += 1000;

}

if (e->midpoint->y == 0) {

my\_offset -= 1000;

}

else if (e->midpoint->y == 100) {

my\_offset += 1000;

}

// draw the actual line

output\_file << "set arrow from ";

output\_file << e->of\_triangle->center->x << ","

<< e->of\_triangle->center->y << " to "

<< (e->midpoint->x+mx\_offset) << ","

<< (e->midpoint->y+my\_offset)

<< " as 2\n";

}

}

current\_e\_node = current\_e\_node->next;

}

current\_p\_node = current\_p\_node->next;

}

// ending of the file:

output\_file << "plot NaN notitle\n";

output\_file << "pause 1000\n"; // ensure that X11 stays visible

output\_file.close();

}

else {

cout << "Error: Could not access output file for writing!\n";

}

}

};

// can be built and run by:

// g++ delaunay.cpp -o delaunay

// ./delaunay <input\_address output\_address <delaunay voronoi>>

int main (int argc, char \* const argv[]) {

if (argc == 1) {

// using default values

// needs the two files in the same directory

// draws voronoi only

Delaunay\* d = new Delaunay("input.txt", "output.txt", false, true);

d->insert\_points();

d->write\_output();

delete d;

}

else if (argc == 3) {

// standard

// uses file addresses provided

// draws voronoi only

string input = argv[1];

string output = argv[2];

Delaunay\* d = new Delaunay(input, output, false, true);

d->insert\_points();

d->write\_output();

delete d;

}

else if (argc == 5) {

// uses file values provided

// uses dealunay and voronoi booleans to draw/not draw delaunay/voronoi

string input = argv[1];

string output = argv[2];

string delaunay\_s = argv[3];

string voronoi\_s = argv[4];

if ((delaunay\_s == "true" || delaunay\_s == "false") &&

(voronoi\_s == "true" || voronoi\_s == "false")) {

// parse boolean values

bool delaunay = (delaunay\_s == "true") ? true : false;

bool voronoi = (voronoi\_s == "true") ? true : false;

Delaunay\* d = new Delaunay(input, output, delaunay, voronoi);

d->insert\_points();

d->write\_output();

delete d;

}

else {

// values provided are not booleans

cout << "Error: Invalid input for delaunay or voronoi argument!\n";

cout << "Make sure delaunay and voronoi arguments booleans\n";

}

}

else {

cout << "Usage: <input output <delaunay voronoi>>\n";

}

return 0;

}