



Université de Pierre et Marie Curie  
Course Project

# Recherche rapide d'un triangle contenant un point dans un maillage

Arne Heimendahl, Olivia Kaufmann

Supervisor:

Advisor:

Submission Date: 13.01.2018

# Contents

<b>1 Project description and aim</b>	<b>2</b>
<b>2 The template class T3 and the class Triangle</b>	<b>2</b>
2.1 The template class T3 . . . . .	2
2.2 The class Triangle . . . . .	2
<b>3 The class Mesh</b>	<b>2</b>
3.1 LoadVertices and LoadTriangles . . . . .	3
3.2 Find the adjacent triangle . . . . .	3
3.2.1 setAdjacencyViaMultimap . . . . .	3
3.2.2 setAdjacencyViaList . . . . .	4
3.2.3 Vergleich . . . . .	5
3.3 L'algorithme promenade . . . . .	5
3.3.1 Random neighbor selection . . . . .	5
3.3.2 Selection of neighbor with minimal oriented volume . . . . .	5
3.4 Can an arbitrary choice of the consecutive triangle be a better choice? . . . . .	5
3.5 Empirical demonstration that the deterministic choice of the consecutive triangle is better in regular meshs . . . . .	6
3.6 The visualization via gnuplot . . . . .	6
<b>4 Cover vertices of a mesh by another mesh</b>	<b>7</b>
<b>5 Test environment- the main function</b>	<b>8</b>

# 1 Project description and aim

The aim of this project is to implement an algorithm which searches for a triangle in a convex mesh covering a point  $(x, y)$ . A linear complexity (in number of vertices of the mesh) is desired.

## 2 The template class T3 and the class Triangle

### 2.1 The template class T3

Objects of the class  $T3$  represent elements of a three dimensional space in  $T$ . Hereby, the data type  $T$  is defined using a template. Within this project, either  $T3 < int >$  for the definition of a triangle (see 2.2) or  $T3 < double >$  for the definition of the coordinates of a vertex (see 3) is used. The private members  $x, y$  and  $z$  of type  $T$  store the entries of the  $T3$  vector. The class has a default constructor, a constructor by copy and a constructor creating a vector given the three vector entries. Moreover, the class has operators to access and modify the entries, add to elements of  $T3$ , multiply an element of  $T3$  by a scalar of the type  $T$  and calculate the scalar product of two  $T3$  vectors. The operator  $<$  compares two elements of  $T3$ ,  $v_1 = (x_1 \quad y_1 \quad z_1)^T$  and  $v_2 = (x_2 \quad y_2 \quad z_2)^T$ , in the following way:

$$v_1 < v_2 \Leftrightarrow \begin{cases} x_1 < x_2 \text{ or} \\ x_1 = x_2 \text{ and } y_1 < y_2 \text{ or} \\ x_1 = x_2 \text{ and } y_1 = y_2 \text{ and } z_1 < z_2 \end{cases} .$$

This allows a lexicographic ordering of a list of  $T3$  elements (see 3.2.2,  $T3 < int >$ ). For three  $T3$  vectors, which define the vertices of a triangle, the method *oriented\_vol* computes the corresponding signed volume. Let  $a, b$  and  $c$  denote  $T3$  vectors. The oriented volume  $A_{abc}$  is defined as

$$A_{abc} = (\vec{ab} \times \vec{bc})^T \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = (b_1 - a_1)(c_2 - b_2) - (b_2 - a_2)(c_1 - b_1)$$

The sign is positive for triangles oriented in trigonometric sense. This method is used in the method *promenade* 3.3 to determine a suitable 'walking' direction.

### 2.2 The class Triangle

The class *Triangle* inherits from the class *T3*. Its derived members  $i, j, k$  are specialized as integers representing the position of its defining vertices in the array of the points of the given mesh. This array is a member of the class *mesh* in 3. In addition, it has three extra integer members *neighbor1*, *neighbor2*, *neighbor3* representing the position of the adjacent triangles in the array of triangles which is also a member of the class *mesh*. Notice that these index lies between zero and the number of triangles minus one.

An adjacent triangle  $t'$  is said to be *neighbor1* of a triangle  $t$  if, saying  $i$  represents the first index of a vertex of  $t$  but  $i$  does not represent a vertex of  $t'$ . The same holds for *neighbor1* and *neighbor2*. Note that this relation is not symmetric, that is to say that  $t'$  is *neighbor1* does not imply that  $t$  is *neighbor1* of  $t'$ . This assignment facilitates the access to the following adjacent triangle in the algorithm *promenade* (see 3.3).

At the creation of a new triangle the latter three members are initialized by  $-1$  which means that a triangle has no neighbors when it is created. The neighbors are set via the functions *setAdjacencyViaMultimap* 3.2.1, respectively *setAdjacencyViaList* 3.2.2. After the execution of one of these two functions  $neighbor1 = -1$  describes the case that there is no adjacent triangle on the opposite side of the first vertex. Since the neighbors are private members there are getter and setter in order to read and manipulate them.

## 3 The class Mesh

The class *Mesh* contains all necessary information of the mesh and the search of a vertex in (or outside) the mesh can be realized by its member functions. Its private members are pointers to the arrays of ver-

tices and triangles of the mesh as integers who store the size of these two lists. In order to create an object of type Mesh the name of the desired *.msh* file has to be transmitted. The file is read by the functions *LoadVertices* and *LoadTriangles* 3.1 who then initialize the members *triangles*, *sommets*, *numbTriangles*, *numbSommets*. Since the members are private there are getter and setter for *numbTriangles* and *numbSommets* as well as getter for *triangles* and *sommets*. The functions *LoadTriangles* and *LoadVertices* are their setters.

l'intersection de 2 triangles distincts soit une arête commune, un sommet commun ou rien, we consider convexe meshes

### 3.1 LoadVertices and LoadTriangles

Both functions are called in the constructor of the class *mesh* setting the members *vertices* and *triangles*. They work basically the same way with the small difference that they create arrays of different data types ( $T3 < double >$  and  $T3 < int >$ ) and searching for different key words in the *.msh* file. At first, the *.msh* file is opened according to its name. Then the functions search for the line indicating the number of vertices respectively the number of triangles. This is implemented by using an *fstream* and the function *getline* who stores the information of a line and skips to the next line.

To make the code work the next line must include the according number which is then stored in *numbVertices*, respectively *numbTri*. The data is read as a string which is transformed to an integer by the command *stoi*. These numbers also define the size of the arrays of type  $T3 < double >$  and *Triangle* which are created by *new* in order to allocate the necessary memory. The following lines must include the coordinates of all vertices, respectively the positions of the triangles in the table of vertices. The lines are read as strings who are casted in to *doubles* respectively *integers*.

Each line defines a vertex or a triangle which is then stores in the correspondent array. Both functions finally return the arrays filled with the vertices or triangles.

### 3.2 Find the adjacent triangle

#### 3.2.1 setAdjacencyViaMultimap

The goal of this function is the initialization of the members *neighbor1*, *neighbor2*, *neighbor3* of the class *Triangle*. As a reminder, they are defined by their position in the array of triangles. Their initialization is realized by the container *multimaps*. Each triangle  $t = (v_{i_1}, v_{i_2}, v_{i_3})$  is represented by the sequence  $(i_1, i_2, i_3)$ , where these indices indicate the position of the vertices  $v_{i_1}$ ,  $v_{i_2}$  and  $v_{i_3}$  in the array of vertices. For each  $t$  three pairs are added to the multimap, where the edges  $(v_{i_k}, v_{i_l})$ ,  $k, l \in \{1, 2, 3\}$ ,  $k \neq l$ , stored by the data type *pair*  $< int, int >$  (more precisely *pair*  $< i_k, i_l >$ ), represent the keys, whereas the mapped value is an integer representing the position of the triangle in the array of triangles. Hence, the initialization if the multimap is realized in  $3n_T$  loops, so in  $O(n_T)$ .

Essentially for the functioning of the method is the ordering of the of the indices  $i_1, i_2, i_3$  in  $t$ . As in *setAdjacencyViaList* (LIST ODER MAPS?), an edge  $\{v_{i_k}, v_{i_l}\}$  has to be uniquely identified by the pair  $(i_k, i_l)$  which should not be mixed up with  $(i_l, i_k)$ . This is achieved by demanding that for each triangle  $t = (v_{i_1}, v_{i_2}, v_{i_3})$  the vertices are ordered in a manner such that  $i_1 < i_2 < i_3$ . Thus, an edge  $(v_{i_k}, v_{i_l})$  can uniquely be identified by the pair  $(i_k, i_l)$ ,  $i_k < i_l$ .

We find the adjacent triangles by the following three steps:

1. We range over the multimap and in each step we save the current key  $(i_k, i_l)$ , its position in the list (this is the current iterator) and the index  $i$  of the triangle  $t = (v_{i_1}, v_{i_2}, v_{i_3})$ . Then we move the iterator to the next element of the list and erase the pair  $((i_k, i_l), i)$  of the multimap which is realized in constant time since the iterator belonging to  $((i_k, i_l), i)$  was saved.
2. Now searching by the function *find* for the stored key  $(i_k, i_l)$  allows us to determine if there is another index  $j$  associated to  $(i_k, i_l)$ . If this is the case (the iterator returned by *find* does not point to the last element of the multimap) the triangles represented by  $i$  and  $j$  are adjacent. The function *find* needs  $O(\log(n_T))$  in order to return an iterator. Note that the pair  $((i_k, i_l), j)$  is not erased. This would yield an error because the actual iterator would try to access that element.

- The indices  $i$  and  $j$  are set as the neighbors of each other where  $j$  is set as neighbor  $k \in \{1, 2, 3\}$  if the index  $k$  does not represent a vertex of the triangle represented by  $j$ . The same rule is applied to  $i$  as a neighbor of  $j$ . This assignment is realized in constant time.

The application of all three steps needs a time of  $O(n_T)O(\log(n_T)) = O(n \log(n_T))$ .

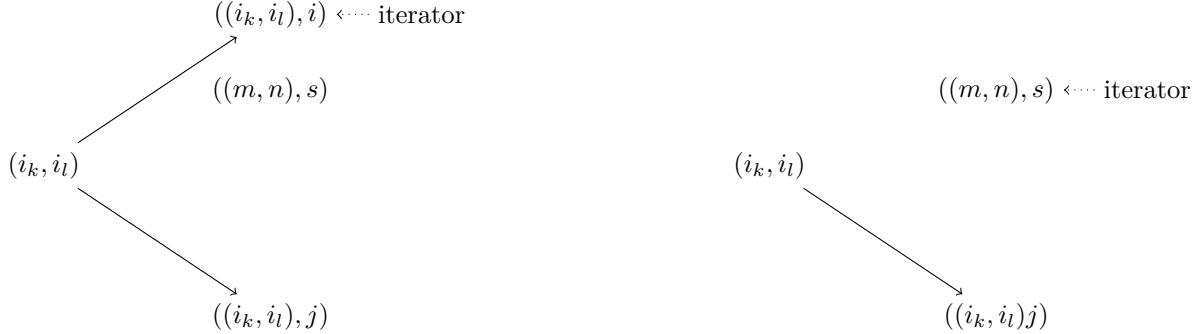


Figure 1: The initial situation for determining the neighbors (left). The element  $((i_k, i_l), i)$  is erased from the map in the iterator points to another element, not necessarily  $((i_k, i_l), j)$ . This essentially depends on the internal ordering of the multimap.

### 3.2.2 setAdjacencyViaList

As in `setAdjacencyViaMultimap`, this method sets the members, describing the neighbors, for each triangle in the considered mesh. In the same manner as before, the members are described using their position in the triangle array. The idea of this method is to create a list in each adjacent triangles appear in consecutive order. After this list is created, the adjacencies of the triangles can be set in linear complexity.

The list with the desired property is created in the following way: For each triangle three new triangles are created and appended to the list. For the triangle  $t = (v_{i_1}, v_{i_2}, v_{i_3})$  with the index  $i$ , a new triangle for every edge is defined. More precisely, the triangles  $(i_1, i_2, i)$ ,  $(i_1, i_3, i)$  and  $(i_2, i_3, i)$  are pushed on the list. The creation of this list has complexity  $O(n_T)$ . In order to archive that two neighboring triangle are positioned one after another in the list, the list is sorted in lexicographic order. XXX The complexity for sorting a list with  $n_T$  elements is  $O(n_T \log n_T)$ .

$$\begin{array}{ccc}
i_1 & i_2 & i \\
i_1 & i_2 & j \\
i_1 & i_3 & i \\
& \vdots & \\
i_1 & * & * \\
i_2 & i_3 & i \\
i_2 & i_3 & k \\
& \vdots & \\
i_{n_T} & * & *
\end{array}$$

keine übereinstimmung bei vergleich , nachbar von der kante schon gesetzt (vorhergehendes listenelement) oder äußere kante

order of edges

After the list creation, the members of the mesh triangles can be set. Due to the lexicographical ordering, the list allows now to define adjacencies between triangles with a number of comparison which is linear in the number of triangles. Two consecutive triangles of the list are considered. The first two entries of the triangles are compared. If they are equal, the triangles share an edge and hence can be defined as adjacent. Particularly, if the triangles with the indices  $i$  and  $j$  have the vertices  $v_k$  and  $v_l$  in common, the triangles  $(k, l, i)$  and  $(k, l, j)$  appear consecutively in the list. The head element of the list is popped from

the front and the comparison is repeated for the next triangles. The comparison of every pair of triangles has constant complexity. As a list of size  $3n_T$  is considered, the complexity of setting the neighbors of the  $n_T$  mesh triangles given this list is  $O(n_T)$ .

In summary, the method `setAdjacencyViaList` has complexity  $O(n_T \log n_T) + O(n_T) = O(n_T \log n_T)$ .

### 3.2.3 Vergleich

Laufzeitvergleich, Komplexität

## 3.3 L'algorithme promenade

Promenade is a method of the class `Mesh` which is given a triangle  $T$  and a point  $p$  of the type  $T3 < double >$ . As output a triangle in the mesh covering  $p$  is returned.

Denote the  $T3 < double >$  vectors of the vertices of the triangle  $T$  by  $c_1, c_2$  and  $c_3$ . Hereby, the vertices are numbered in trigonometric sense. Thus, the oriented volume of the triangle  $(c_1, c_2, c_3)$  is positive. Recall that `oriented_volume` is a method of the class `T3` which takes two additional `T3` vectors as input (see 2.1). (negierung in code) For each edge of the triangle  $T$ , a new triangle containing the edge and the point  $p$  is considered. We are interested in the oriented volumes of the triangles  $(c_1, c_2, p), (c_2, c_3, p)$  and  $(c_3, c_1, p)$ . If the oriented volume of one of those new triangles is positive, then  $p$  and the triangle  $T$  are contained in the half plane defined by the currently considered edge of  $T$ . Consequently,  $p$  is contained in  $T$  if the oriented volumes of the triangles  $(c_1, c_2, p), (c_2, c_3, p)$  and  $(c_3, c_1, p)$  are all positive. In this case, the method `promenade` returns the input triangle  $T$ . On the other hand, if there is one triangle with negative oriented volume,  $T$  is not covering  $p$ . We choose a neighboring triangle  $N$  of  $T$  with negative oriented volume and call the method `promenade` recursively with starting triangle  $N$ .

the oriented volume of crossed edge is now positive, edge in other direction to consider neighboring triangle in trigonometric sense

head in direction of  $p$ , approach

pay attention for triangles on the border of the mesh. neighbor initialization with -1, just 2 neighboring triangles

Optionally, the method `promenade` takes also a vector with entries of the class `Triangle`, called path, as input variable. The current triangle is pushed back on the vector and, due to the recursion, the vector stores the consecutively passed triangle when the algorithm terminates. The path is used to visualize the results of the method `promenade` in ??.

terminates because?

### 3.3.1 Random neighbor selection

`random_neg`

random number generation

### 3.3.2 Selection of neighbor with minimal oriented volume

`min_neg`

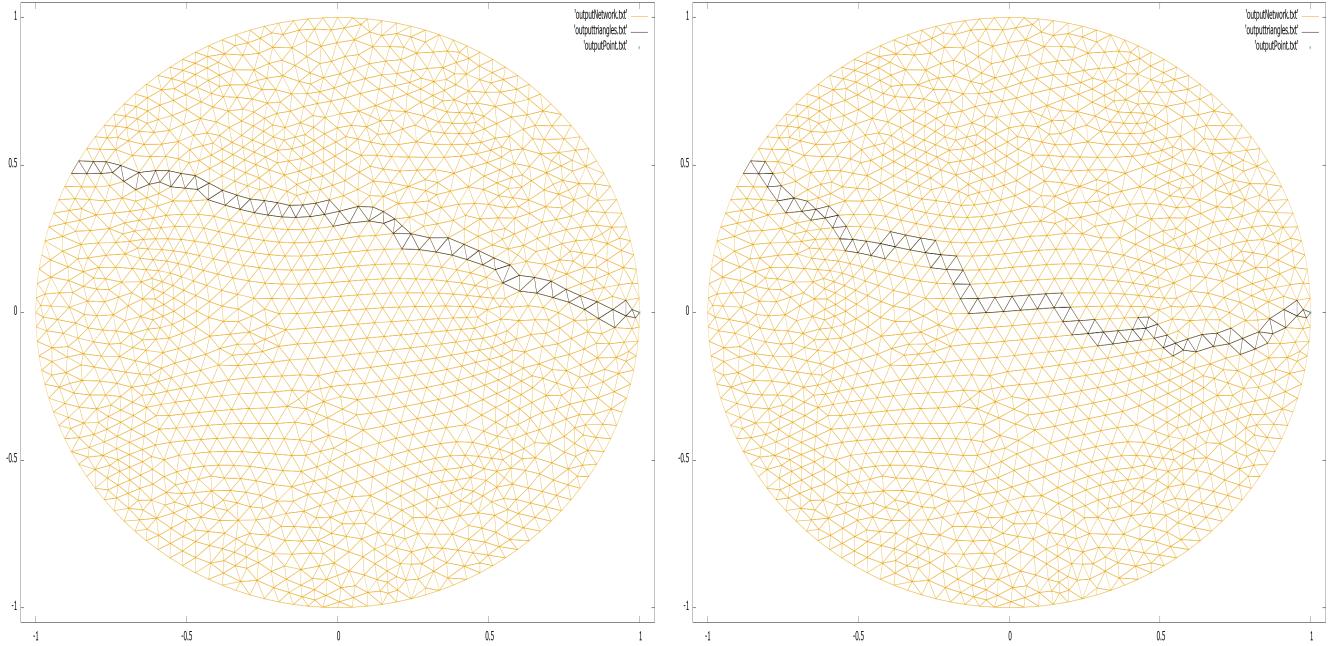
same path length for multiply execution with same starting triangle and point  $p$  in most cases  
only case in which neighbor selection is not uniquely determined:  $p$  on half plane

## 3.4 Can an arbitrary choice of the consecutive triangle be a better choice?

This example shows that a random choice of the consecutive triangle can be better. Assume that the algorithm `promenade` currently considers the triangle  $A$  of figure 3.4 and searches for a covering triangle of the point  $p$ . Hence, the algorithm will proceed with triangle  $B$  or  $C$ .

If we use the size of the oriented volume (the oriented volume of the convex hull of  $p$  and the lower right edge of  $A$ , respectively the upper right edge of  $A$ ) we will clearly choose  $C$ , as illustrated in the second figure (the green area is clearly bigger than the blue one). This is obviously the worse choice since  $B$  is a covering triangle of the point  $p$ .

An important observation is that the regularity of the triangles plays an important role. In this case,



regularity means that all triangles of the mesh have a 'similar' area, 'similar' angles and its vertices are adjacent to a 'similar' number of triangles.

### 3.5 Empirical demonstration that the deterministic choice of the consecutive triangle is better in regular meshes

If we take a look at figures 3.5 and 3.5 it becomes empirically clear that the minimum choice of the consecutive triangle speeds up the algorithm. Figure 3.5 even states that the path created by the minimum choice is always shorter or as long as the path created by the random choice. Looking at the runtime in 3.5 we see that the runtime is about as twice as high for the random choice when the starting triangle and the given point have a big distance.

Nevertheless, it has to be kept in mind that the underlying mesh plays an important role. In this case the mesh 'maillage5.msh' is regular in the sense explained in the previous section.

### 3.6 The visualization via gnuplot

The visualization is done by the function *exportGnuplot* being able to visualize two different things:

1. The result of the algorithm *promenade*, that is a sequence of adjacent triangles from a starting triangle to a triangle covering the searched point. In this case the input variable *triangles\_path* contains the sequence of triangles and the input variable *points* contains just one point, hence the input *intnumpoints* is 1.
2. Given a mesh and the vertices of another mesh it can visualize the set of the covering triangles (a subset of triangles of the first mesh) of the points of the second mesh. In this case the input variable *triangles\_path* contains the covering triangles and the *points* (IN VERTICES UMBENENNEN?, würde ich nicht, denn die variable heißt points und in diesem Fall werden sie auch nur als points und nicht als vertices behandelt) are the vertices of the second mesh.

The functions writes four text files, one for all triangles of the mesh, one for the triangles in *triangles\_path*, one for the points in *points* and one for the commands which shall be executed by gnuplot. This is all realized by a simple *ofstream* variable, allowing to create and manipulate a text file.

The script for gnuplot contains a line which keeps the plot open until some key is hit in the terminal.

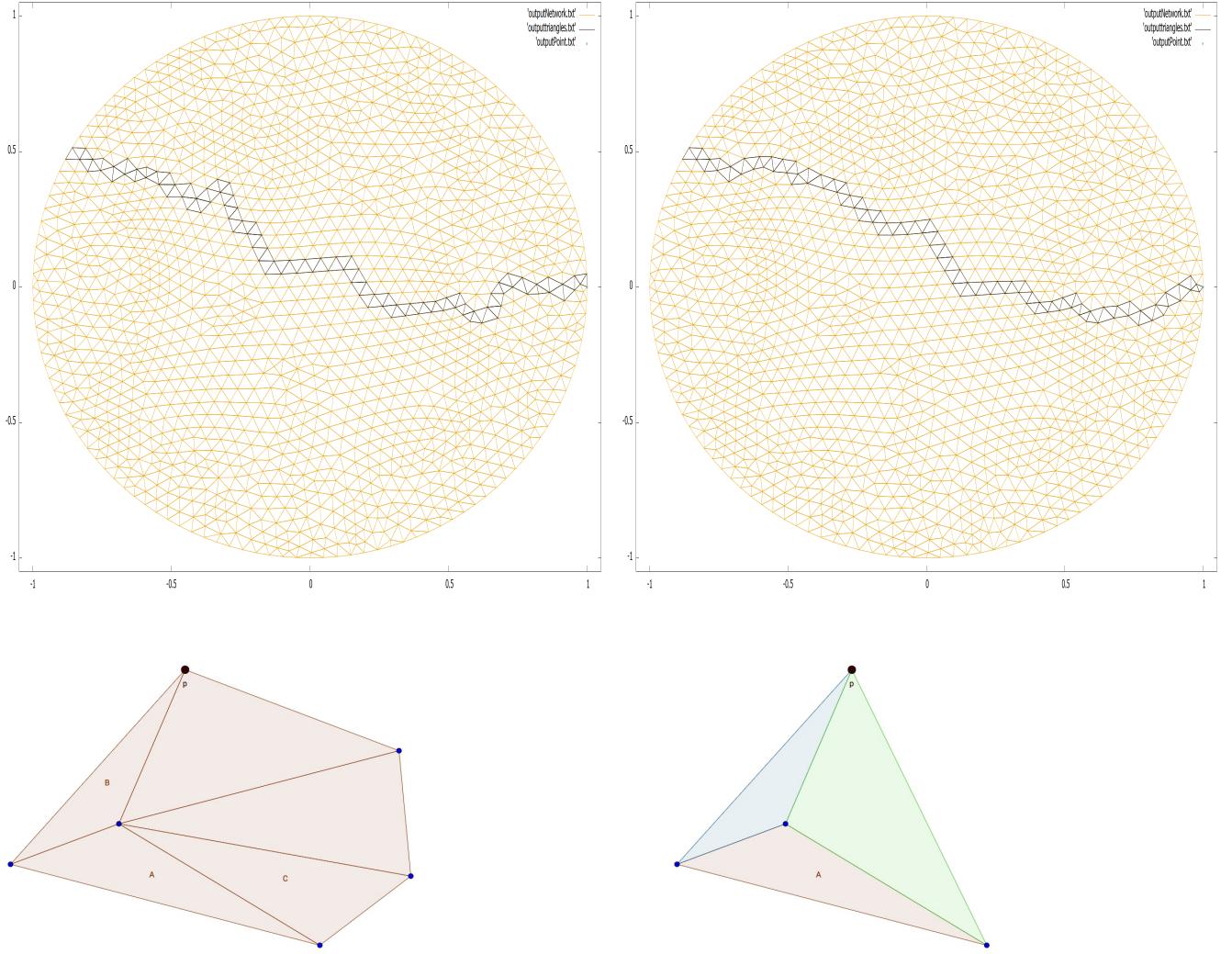


Figure 2: The illustration of 3.4.

The actual execution in the terminal is achieved by the command *system* which executes its input string in the terminal.

## 4 Cover vertices of a mesh by another mesh

The aim of this section is to find the covering triangle for every vertex of another mesh in the same domain. This can be useful when a mesh is refined or coarsened. For this purpose, the promenade algorithm is executed frequently, once for every vertex. In order nevertheless to obtain a good running time, the starting triangle for *promenade* should be chosen in an efficient way. The idea is to use starting triangle for which we already know that they are located close to the vertex to cover.

The function *findVertices* is given two meshes  $m$  and  $M$  and a reference of a vector of triangles *coveringTriangle* and returns *coveringTriangle* after modification. The function's goal is to search for covering triangles in  $M$  for the vertices of  $m$ . At the position  $i$  of *coveringTriangles* the triangle in  $M$  covering the  $i$ -th vertex of the mesh  $m$  is stored.

First, two random triangles, one in  $m$  and one in  $M$ , are chosen. In *findVertices* covering triangles for the vertices of the triangle randomly chosen in  $m$  are computed.

start with random triangle of mesh m (JUST START WITH FIRST TRI?)

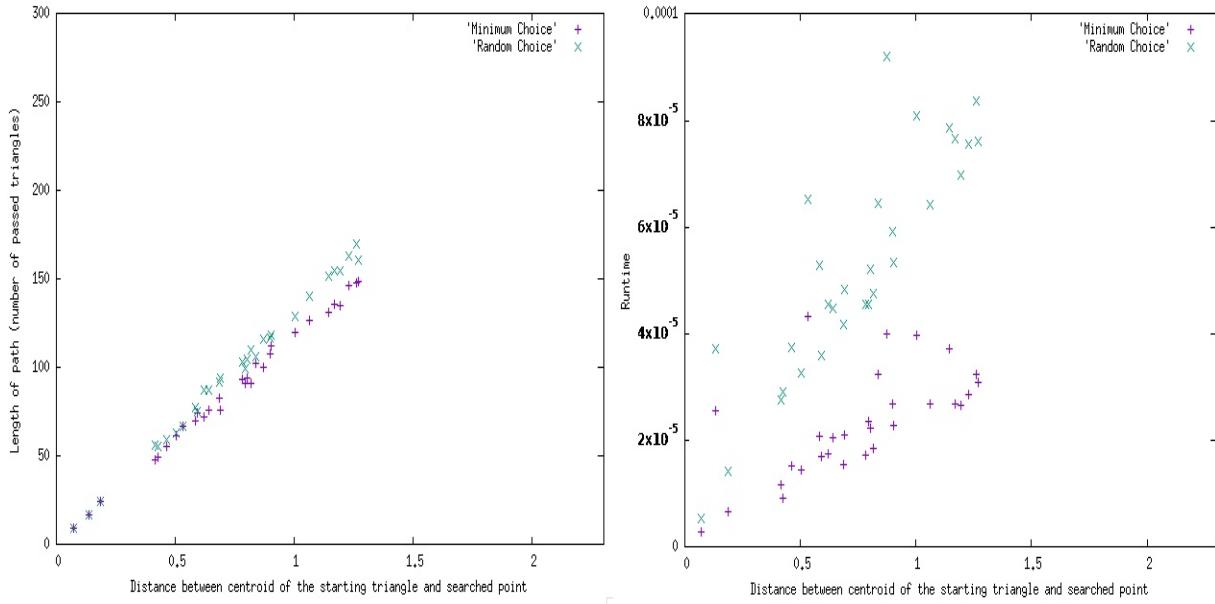


Figure 3: The pathlength and the runtime compared using  $\min_{neg}$  and  $random_{neg}$ . In this example, 30 data points were randomly created in the file 'TimeMeasurements.cpp'. The underlying mesh is 'maillage5.msh'.

random triangle in mesh  $M$ , use as a starting triangle for *promenade*, find covering triangle in  $M$  for the vertex XX call *promenade* (not that we do not store the path taken by *promenade* here)  
 covering triangle: as triangles initialized as with vertex with index 0 (start at 1 in mesh files)  
 choose triangle in  $m$ , cover all vertices of this triangle (use covering tri in  $M$  for neighboring vertex as starting tri)

#### RECURRENCE

covers vertices of one triangle and calls *TriangleRecurrence*, pass the two meshes, the vector covering-Triangles and the triangle in  $m$  which we started with and which is already covered  
 recurrence modifies the vector of coveringTriangles

search for covering triangles for the vertices of the neighboring triangles of the already covered triangle, the covering triangles are 'relatively' close, so we hope to have a fast termination of the algorithm *promenade* like this

start with member neighbor1 then 2 3 (have to check for existence of the neighbors)

searching structure is similar to approach of depth-first search

*selectAdjacentPoint* to select Triangle to start *promenade* (if the vertex is not already covered), then cover neighbors of this triangle

## 5 Test environment- the main function

The cpp file main offers the possibility to test the properties of the program:

1. Find the neighbors of a triangle.
2. Find the triangle for a point (the point can be chosen).
3. Find the covering triangles for the vertices of another mesh.

All three options contain the option to choose the mesh where the mesh becomes finer from maillage1.msh increasingly to maillage5.msh. Of course it is also possible to add own meshes. In this case yet, it has to be noted that the triangle's indices ordering must be increasing, as mentioned in the sections before. Moreover, the shown intervall is set

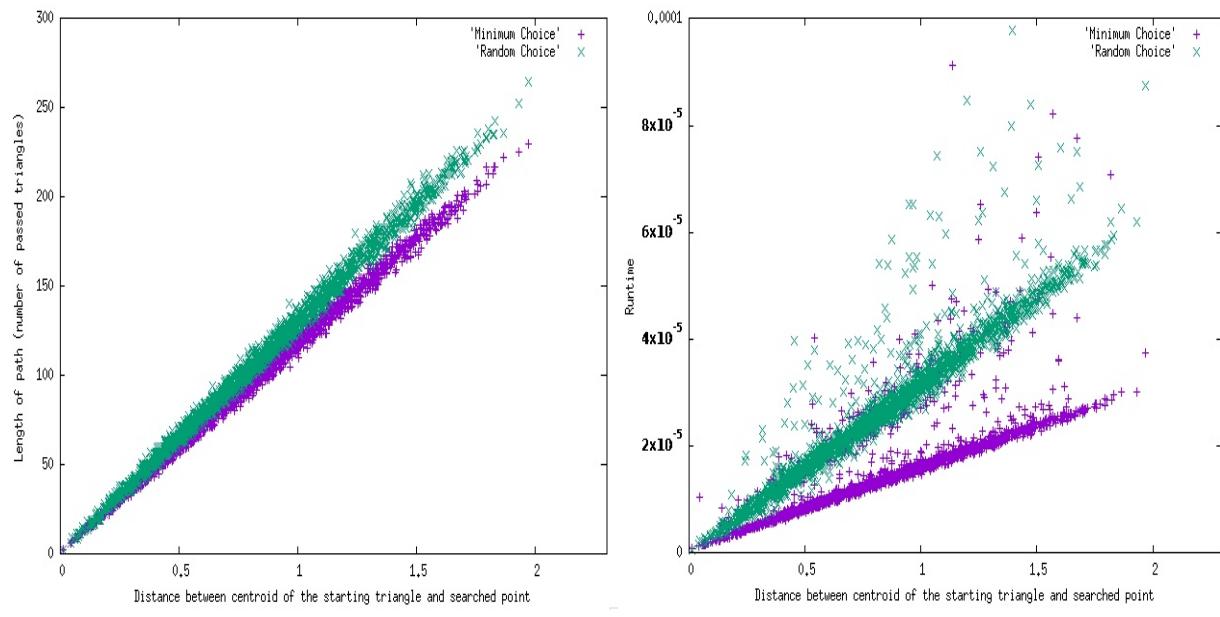


Figure 4: Here, 2000 data points were randomly created.