Delaunay Triangulation

Daniel VandenHeuvel

December 1, 2022

Contents

Chapter 1

Delaunay Triangulation

This chapter discusses Delaunay triangulations and their implementation. To get familiar with the idea, we initially implement the clearer algorithm of de Berg et al. (2008), and then later we consider those ideas discussed by the more recent book of Cheng et al. (2013). To implement these algorithms, we will need to first discuss some data structures.

1.1 Directed Acyclic Graphs

We start with a discussion of graphs and, in particular, directed acyclic graphs. We start by making some definitions, following the descriptions provided by Cormen et al. (2022, Appendix B.4) and Deo (2018) and Mehta (2018).

Definition 1.1 (Directed graph)**.** Let $G = (V, E)$ be some graph with *vertex set* V and *edge set* E. This graph is a *directed graph* if the elements of E, called *edges*, are ordered pairs rather than unordered pairs, i.e. $(u, v) \in E$ and $(v, u) \in E$ represent different edges in the graph, where $u, v \in V$. We may call a directed graph a *digraph* for short. ■

Definition 1.2 (Acyclic graph)**.** Let $G = (V, E)$ be a directed graph. A sequence $(v_0, v_1, \ldots, v_k)$ of vertices with $u = v_0$, $u' = v_k$, and $(v_{i-1}, v_i) \in E$ for $i = 1, \ldots, k$ is called a *path* of length k from the vertex u to the vertex u'. If $v_0 = v_k$ and the path contains at least one edge, then we call the path a *cycle*, and the cycle is *simple* if the vertices $v_1, \ldots, v_k$ are distinct. If G contains no simple cycles, then the graph is *acyclic*. ■

Note that the restriction to simple paths in the above definition for an acyclic graph does not permit non-simple cycles, since any directed graph with a cycle necessary contains a simple cycle. For example, consider the cycle $C = (2, 3, 6, 9, 11, 12, 6, 3, 2)$. This is not simple since the vertices are not all unique after $v_0 = 2$, but we can take a subset of this cycle between an element and its next occurrence, such as $C' = (6, 9, 11, 12, 6)$, to obtain a simple cycle.

Definition 1.3 (Directed acyclic graph)**.** A *directed acyclic graph*, or *DAG*, is a graph $G = (V, E)$ is a directed graph that is acyclic.

1. If we define the *out-degree* of a given node $v \in V$ to be the number of edges leaving v, i.e. the number of elements in the set $\{u \in V : (v, u) \in E\}$, and the *in-degree* to similarly be the number of elements in the set $\{u \in V : (u, v) \in E\}$, then the *degree* of v is defined to be its in-degree plus its out-degree. The degree of v is denoted $\deg v$.

2. If a node $v \in V$ has degree 0, it is called a *leaf node*.

3. The *children* of a node v are the nodes that v connects to, i.e. the set $\{u \in V : (v, u) \in E\}$.

4. If the DAG has a single *root*, meaning a unique node from which every other node can be reached (also called the upper-most node), then we may called the DAG a *rooted DAG*. ■

For our application, we are primarily interested in the use of DAGs for point location. We will return to DAGs once we have introduced the Delaunay triangulation, and make their importance clear. We use `SimpleGraphs.jl` (Scheinerman, 2014) to implement DAGs in JULIA (Bezanson et al., 2017).

1.2 Triangulations

To introduce the Delaunay triangulation, we need to first introduce triangulations. We follow the description given by de Berg et al. (2008).

Definition 1.4 (Maximal planar subdivision)**.** A *maximal planar subdivision* is a subdivision of the plane together with some vertex set V and edge set E such that no edge connecting two vertices can be added without destroying the subdivision's planarity. ■

Definition 1.5 (Triangulation)**.** Let $P = \{p_1, \ldots, p_n\}$ be a set of points in the plane. A *triangulation* of P is a maximal planar subidivision whose vertex set is P. ■

An important measure to keep track of in a triangulation is the angle-vector.

Definition 1.6 (Angle-vector)**.** Let $\mathcal{T}$ be a triangulation of P, and suppose it has m triangles. Sort the $3m$ angles of the triangles of $\mathcal{T}$ in ascending order, so that $\alpha_1 \cdots, \alpha_{3m}$ denotes the resulting sequence of angles with $\alpha_i \leq \alpha_j$ for $i < j$. The *angle-vector* of $\mathcal{T}$ is denoted $A(\mathcal{T}) = (\alpha_1, \ldots, \alpha_{3m})$. If we have two triangulations $\mathcal{T}$ and $\mathcal{T}'$ of P, then we say that the angle-vector of $\mathcal{T}$ is larger than the angle-vector of $\mathcal{T}'$ if $A(\mathcal{T})$ is lexicographically larger[1] than $A(\mathcal{T}')$, or, in other words, if there exists an $i \in \{1, \ldots, 3m\}$ such that

$$\alpha_j = \alpha_j', \text{ for all } j < i, \quad \text{and} \quad \alpha_i > \alpha_i'.$$

In this case, we write $A(\mathcal{T}) > A(\mathcal{T}')$. ■

We are interested in finding the triangulation which maximises the smallest angle, meaning one that maximises the angle-vector.

Definition 1.7 (Angle-optimal triangulation)**.** A triangulation $\mathcal{T}$ of a set P is called angle-optimal if $A(\mathcal{T}) \geq A(\mathcal{T}')$ for all triangulations $\mathcal{T}'$ of P. ■

A fundamental concept in the development of Delaunay triangulation algorithms is the idea of an illegal edge, with the idea that we can "flip" edges in a triangulation to continually increase the angle-vector. Such a flip is called a *Lawson flip* (Cheng et al., 2013). The edge is illegal if we can increase the smallest angle, locally, by flipping that edge.

[1] A vector $\mathbf{x} \in \mathbb{R}^n \setminus \{\mathbf{0}\}$ is said to be lexicographically positive if there is some index $i \in \{1, \ldots, n\}$ such that $x_j = 0$ for $j < i$, and $x_i > 0$. We say that $\mathbf{x}$ is lexicographically greater than $\mathbf{y}$ if $\mathbf{x} - \mathbf{y}$ is lexicographically positive. This is a total ordering on vectors.

Definition 1.8 (Illegal edge)**.** Suppose we have a triangulation $\mathcal{T}$ of P, and let $e = \overline{p_i p_j}$ be an edge of $\mathcal{T}$. This edge will be incident to two triangles $p_i p_j p_k$ and $p_i p_j p_\ell$.[2] Provided these two triangles form a convex quadrilateral, we can obtain a new triangulation $\mathcal{T}'$ by removing $\overline{p_i p_j}$ from $\mathcal{T}$ and inserting $\overline{p_k p_\ell}$ instead. This is called an *edge flip*. If $\alpha_1, \ldots, \alpha_6$ are the six angles defined by the two triangles, and $\alpha'_1, \ldots, \alpha'_6$ are those by the new triangles in $\mathcal{T}'$, then these are the only differences between $A(\mathcal{T})$ and $A(\mathcal{T}')$; see Figure 1.1. If

$$\min_{i=1}^{6} \alpha_i < \min_{i=1}^{6} \alpha'_i,$$

then we call the edge $e = \overline{p_i p_j}$ an *illegal edge*. ■

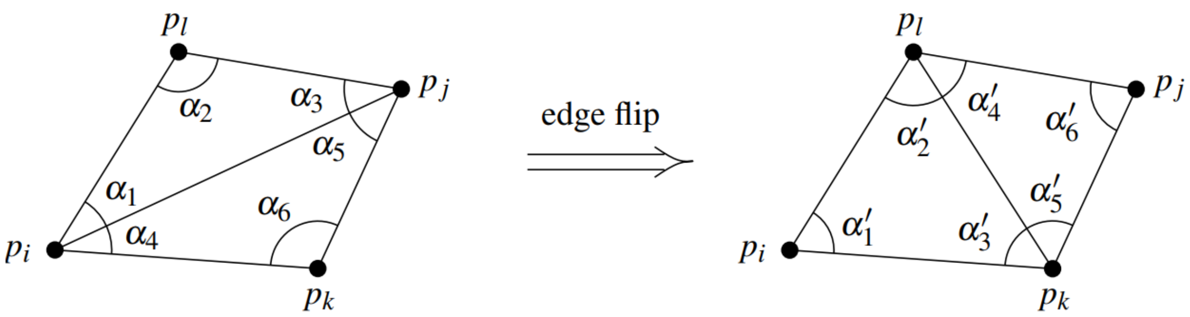
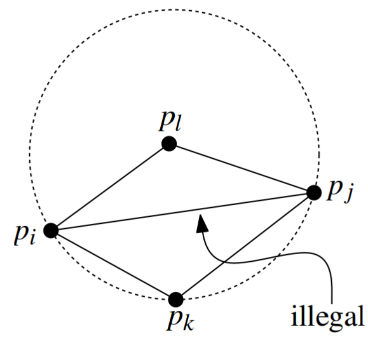


Figure 1.1: Flipping an edge.

The next lemma will give us a nice test later for checking whether a triangulation is a Delaunay triangulation, without having to compute any angles.

Lemma 1.1 (Illegal edge criterion). Let edge $\overline{p_i p_j}$ be incident to triangles $p_i p_j p_k$ and $p_i p_j p_\ell$, and let C be the circle through p_i, p_j, and p_k. The edge $\overline{p_i p_j}$ is illegal if and only if the point p_ℓ lies in the interior of C. Furthermore, if the points p_i, p_j, p_k, and p_ℓ form a convex quadrilateral and do not lie on a common circle, then exactly one of $\overline{p_i p_j}$ and $\overline{p_k p_\ell}$ is an illegal edge. ■



Definition 1.9 (Legal triangulation)**.** A *legal triangulation* is a triangulation with no illegal edges. ■

Since legalising an edge increases the angle-vector, any angle-optimal triangulation cannot have any illegal edges. In particular, any angle-optimal triangulation is a legal triangulation.

1.3 The Delaunay Triangulation

Now let us introduce the Delaunay triangulation. Some of this material borrows from Cheng et al. (2013), avoiding some of the links with Voronoi tessellations made in the description of Delaunay triangulations by de Berg et al. (2008). We assume that the

[2]Unless e is an edge of the convex hull of P, in which case it is incident to only a single triangle.

points in our point sets are in *general position*, meaning no four points in the point set are on a circle.

Definition 1.10 (Delaunay triangulation)**.** Let P be a finite point set. A triangle $p_ip_jp_k$ is *Delaunay* if $p_i, p_j, p_k \in S$ and its open circumcircle, i.e. the interior of the circle through p_i, p_j, p_k, contains no points in P. An edge is *Delaunay* if its vertices are in P and it has at least one empty open circumcircle. A *Delaunay triangulation* of P, denoted $\mathcal{DT}(P)$, is a triangulation of P in which every triangle is Delaunay.

We note that this definition is equivalent to the definition in de Berg et al. (2008), thanks to de Berg et al. (2008, Theorem 9.7). Remarkably, this open circumcircle property is enough to give the following theorem.

Theorem 1.1 (Legal triangulations are Delaunay)**.** *Let P be a set of points in the plane. A triangulation $\mathcal{T}$ of P is legal if and only if $\mathcal{T} = \mathcal{DT}(\mathcal{P})$.*

Not only are all legal triangulations Delaunay triangulations, but the Delaunay triangulation is the triangulation that maximises the minimum angle.

Theorem 1.2 (Delaunay triangulations maximise the angle-vector)**.** *Let P be a set of points in the plane. Any angle-optimal triangulation of P is a Delaunay triangulation of P. Furthermore, any Delaunay triangulation of P maximises the minimum angle over all triangulations of P.*

1.4 Data Structures for Delaunay Triangulations

We maintain a set of data structures for our Delaunay triangulations. There may be some differences across different algorithms, but here we will describe the set of data structures common to all. To assist in our discussion, we use the triangulation in Figure 1.3 as an example. In what follows, triangles are represented as tuples $T = (i, j, k)$, abbreviated as T_{ijk}, and T is treated as being equivalent to (j, k, i) and (k, i, j). In these tuples, the indices refer to the index of the point, e.g. T_{ijk} means the triangle going from p_i to p_j to p_k, in that order. All triangles are treated as being positively oriented. Points are treated as tuples of coordinates, and the ith point will be denoted $p_i = (x_i, y_i)$. Collections of triangles will be sets, meaning a hash map that stores only the keys rather than any values, and collections of points will be vectors. Edges are treated as tuples (i, j), abbreviated e_{ij}, and collections of edges will also be sets. Just like with triangles, e_{ij} refers to the edge from p_i to p_j (with orientation). For example, the set of triangles in Figure 1.3 is $\mathcal{T} = \{T_{154}, T_{135}, T_{146}, T_{163}, T_{362}, T_{325}\}$, the set of points is $\mathcal{P} = [p_1, p_2, p_3, p_4, p_5, p_6]$, and e.g. the edges of T_{154} are e_{15}, e_{54}, and e_{41}. We may also denote by $\mathcal{V}$ the set of vertices, e.g. in Figure 1.3 we have $\mathcal{V} = \{1, 2, 3, 4, 5, 6\}$.

1.4.1 The adjacent map

It turns out to be important to find, given some oriented edge (u, v), a vertex w for which $(u, v, w) \in \mathcal{T}$, with $\mathcal{T}$ denoting the set of triangles, is a positively oriented triangle. For example, in Figure 1.3 the vertex w associated with the edge $(3, 2)$ is $w = 5$, as $(3, 2, 5)$ is positively oriented. Note that the edge in the other direction, $(2, 3)$, would be associated with $w = 6$ instead. We define an adjacent map $\mathcal{A}$ such that $\mathcal{A}(u, v) = w$, where (u, v, w) is a positively oriented triangle in $\mathcal{T}$, or $\mathcal{A}(u, v) = \partial$ whenever (u, v) is

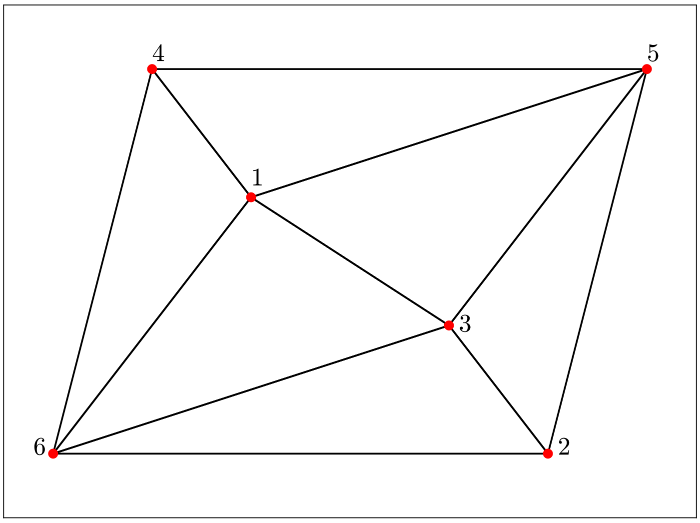


Figure 1.3: An example triangulation.

either a boundary edge. If (u, v) is not an edge in the triangulation, we return $\mathcal{A}(u, v) = \emptyset$. For example, $\mathcal{A}(6, 2) = 3$ but $\mathcal{A}(2, 6) = \partial$ in Figure 1.3, and $\mathcal{A}(4, 2) = \emptyset$. We may also write $\mathcal{A}(e_{uv}) = \mathcal{A}(u, v)$.

We represent adjacent maps as dictionaries mapping edges to vertices, being careful with inserting and deleting edges as we update our triangulations. To accommodate returning $\emptyset$ whenever (u, v) is not an edge, we use what is known as a `DefaultDict`, a dictionary with an extra feature that returns a default value $\emptyset$ whenever it is called at a key that does not exist. We use **`DataStructures.jl`**'s implementation of the `DefaultDict` (Lin, 2013).

1.4.2 The adjacent-to-vertex map

A second data structure, less important than the adjacent map, is a map which takes vertices w to all edges (u, v) such that (u, v, w) is a positively oriented triangle. In particular, w maps to all (u, v) for which $\mathcal{A}(u, v) = w$. This is called the adjacent-to-vertex map, and motivated by the above relationship with the adjacent map we denote it by $\mathcal{A}^{-1}$ so that

$$\mathcal{A}^{-1}(w) = \{(u, v) : \mathcal{A}(u, v) = w\}.$$

For example, in Figure 1.3, we have $\mathcal{A}^{-1}(1) = \{(5, 4), (4, 6), (6, 3), (3, 5)\}$ and $\mathcal{A}^{-1}(5) = \{(4, 1), (1, 3), (3, 2)\}$. Notice that we can also use this map to obtain the boundary of the triangulation, using the fact that $\mathcal{A}(u, v) = \partial$ whenever (u, v) is a boundary edge. In Figure 1.3, we find that $\mathcal{A}^{-1}(\partial) = \{(5, 2), (2, 6), (6, 4), (4, 5)\}$. We may also write $\mathcal{A}^{-1}(w) = \{(e_{uv} : \mathcal{A}(e_{uv}) = w\}$.

We represent the adjacent-to-vertex map as a dictionary that maps a vertex to a set of edges, again being careful with inserting and deleting edges as we update our triangulations, and being especially careful of tracking boundary edges so that we can easily obtain the boundary later.

1.4.3 Graph representation of a triangulation

When we have a triangulation, we may need to know the neighbours of a point in the triangulation, where we define the neighbourhood $\mathcal{N}(i)$ of a point i to be the set of all points j such that (i, j) is an edge in the triangulation. For example, $\mathcal{N}(1) = \{4, 5, 6, 3\}$ in Figure 1.3 and $\mathcal{N}(6) = \{4, 1, 3, 2\}$. This is an example of an undirected graph, and to implement it we will use `SimpleGraphs.jl` (Scheinerman, 2014).

1.5 Operations on Delaunay Triangulations

The methods we use for computing Delaunay triangulations rely on several individual operations. To simplify the discussion, we will start by discussing these operations first, and then later we will put everything together to discuss actual algorithms for computing Delaunay triangulations. Note also that some of these algorithms may be updated slightly later for specific algorithms, e.g. Algorithm 16 is later modified to Algorithm 22 when discussing the Bowyer-Watson algorithm in Section 1.6.2.

1.5.1 Useful subroutines

We list below some common subroutines that we will need. We will frequently need to test if an edge is on the boundary, and so Algorithm 1 is used to test this.

Algorithm 1 Testing if an edge is a boundary edge.

Inputs:
- An edge e_{ij} to test and an adjacent map $\mathcal{A}$.

Outputs:
- Returns `true` if e_{ij} is a boundary edge, and `false` otherwise.

function IsBoundaryEdge(i, j, $\mathcal{A}$) **return** $\mathcal{A}(e_{ij}) == \partial$
end function

We will also often need to rotate a triangle based on some Boolean values. This is useful for example if we have an edge e_{uv} that we want to perform some function on, and we have a Boolean telling us what edge this is. Instead of writing different functions for each possible edge, we can just rotate the triangle into a standard configuration so that the first two vertices give this edge e_{uv}. Algorithm 2 does this rotation.

Another task that we often need to perform is that of checking if an edge actually exists in the triangulation. This test is done in Algorithm 3.

A key predicate needed in computing triangulations is that of testing whether a point is in a given circle. We define this in Algorithm 4. The definition of this predicate in Algorithm 4 is that given by Cheng et al. (2013, Eq. 3.4), although we compute the sign of the given determinant exactly using `ExactPredicates.jl` (Lairez, 2019).

We use a predicate `IsInTriangle` in Algorithm 5 to determine if a point is inside a given triangle T_{ijk}. In this predicate, we decide if a point is inside T_{ijk} by seeing if it is to the left of all the edges e_{ij}, e_{jk}, and e_{ki} of T_{ijk} (noting that T_{ijk} is positively oriented). If it is to the left of all these edges, then indeed p_r is inside T_{ijk} or on one of the edges. The predicate we use for determining if a point p_r is to the left of an edge e_{ij} is given by `IsLeftOfLine` in Algorithm 7 below, making use of `IsOriented` in Algorithm 6. This predicate first considers the cases for the edges on the super triangle (defined in de Berg's

Algorithm 2 Rotating a triangle based on Boolean values.

Inputs:

- A triangle T_{ijk} to rotate.
- Booleans b_{ij}, b_{jk}, and b_{ki} such that only one is true.

Outputs:

- The true Boolean's subscripts define the first two vertices, e.g. if b_{jk} is the true value then the function returns T_{jki}.

```
function RotateTriangle(b_ij, b_jk, b_ki, i, j, k)
    b_ij && return T_ijk
    b_jk && return T_jki
    b_ki && return T_kij
end function
```

Algorithm 3 Testing if an edge exists.

Inputs:

- An edge e_{ij} to test and an adjacent map $\mathcal{A}$.

Outputs:

- Returns `true` if e_{ij} exists, and `false` otherwise.

function EdgeExists(i, j, $\mathcal{A}$) **return** $\mathcal{A}(e_{ij}) \neq \emptyset$
end function

Algorithm 4 Testing if a point is in a circle.

Inputs:

- A point p_ℓ.
- A point set $\mathcal{P}$ and points p_i, p_j, p_k.

Outputs:

- Returns 1 if p_ℓ is in the circle C_{ijk} through p_i, p_j, and p_k, 0 if p_ℓ is on C_{ijk}, and -1 if p_ℓ is outside C_{ijk}.

function IsInCircle($\mathcal{P}$, i, j, k, ℓ)
    $a_x, a_y = \mathcal{P}(i)$ ▷ $\mathcal{P}(i)$ returns the ith point p_i in the point set $\mathcal{P}$,
    $b_x, b_y = \mathcal{P}(j)$ ▷ and $a_x, a_y = \mathcal{P}(i)$ returns the x- and y-coordinates of p_i.
    $c_x, c_y = \mathcal{P}(k)$
    $d_x, d_y = \mathcal{P}(\ell)$
    $$\Delta = \begin{vmatrix} a_x - d_x & a_y - d_y & (a_x - d_x)^2 + (a_y - d_y)^2 \\ b_x - d_x & b_y - d_y & (b_x - d_x)^2 + (b_y - d_y)^2 \\ c_x - d_x & c_y - d_y & (c_x - d_x)^2 + (c_y - d_y)^2 \end{vmatrix}$$
    return $\operatorname{sgn}(\Delta)$
end function

method, Section 1.6.1), and then computes a determinant Δ that is given by Cheng et al. (2013, Equation 3.2). $\operatorname{sgn}(\Delta) = 1$ means the point is to the left, $\operatorname{sgn}(\Delta) = 0$ means the point is on the line, and $\operatorname{sgn}(\Delta) = -1$ means the point is to the right. We compute the sign of Δ exactly using `ExactPredicates.jl` (Lairez, 2019).

If we know that a point is on the edge of a triangle, but we do not what edge it is on, we use Algorithm 8 to return this edge.

Algorithm 5 Testing if a point is in a triangle.

Inputs:

- A triangle T_{ijk}.
- A point set $\mathcal{P}$.
- A query point p_r.

Outputs:

- Returns 1 if p_r is inside T_{ijk}, 0 if p_r is on T_{ijk}, and -1 if p_r is outside T_{ijk}.

function $\textsc{IsInTriangle}(i, j, k, \mathcal{P}, r)$
  $(T_{ijk} == T_{-1,-2,-3})$ `&&` **return** 1 ▷ All points are inside the super triangle.
  $\ell_{ij} =$ `IsLeftOfLine`$(\mathcal{P}, i, j, r)$
  $\ell_{jk} =$ `IsLeftOfLine`$(\mathcal{P}, j, k, r)$
  $\ell_{ki} =$ `IsLeftOfLine`$(\mathcal{P}, k, i, r)$
  $(\ell_{ij} == 0$ `||` $\ell_{jk} == 0$ `||` $\ell_{ki} == 0)$ `&&` **return** 0 ▷ Point is on an edge.
  $(\ell_{ij} == 1$ `&&` $\ell_{jk} == 1$ `&&` $\ell_{ki} == 1)$ `&&` **return** 1 ▷ Point is in the interior.
  return -1 ▷ Point is in the exterior.
end function

Algorithm 6 Orientation of points.

Inputs:

- Three points a, b, c.

Outputs:

- Returns 1 if the points are a, b, c are positively oriented, -1 if negatively oriented, and 0 if the points are collinear. Alternatively, returns 1 if c is left of the line $\overrightarrow{ab}$, -1 if c is right of $\overrightarrow{ab}$, and 0 if the points are collinear.

function $\textsc{IsOriented}(a, b, c)$
  $a_x, a_y = a$
  $b_x, b_y = b$
  $c_x, c_y = c$
  $\Delta = \begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix}$
  return $\operatorname{sgn}(\Delta)$
end function

1.5.2 Adding a triangle

Let us first discuss the problem of adding a triangle into an existing triangulation. We discuss this by way of example. Figure 1.4 shows a series of (not necessarily Delaunay) triangulations. Figure 1.4a is the initial triangulation, Figure 1.4b shows the triangulation with a new triangle added into the interior, Figure 1.4c adds a new triangle onto a single boundary edge, and Figure 1.4d adds a new triangle onto two boundary edges. The goal in this section is to describe the procedure for adding these new triangles, and in particular the updating of the data structures in response to these new triangles, especially for dealing with the boundary edges.

Let us first discuss the addition of the triangle T_{137} into the triangulation. We would first update the adjacent map $\mathcal{A}$ so that $\mathcal{A}(e_{13}) = 7$, $\mathcal{A}(e_{37}) = 1$, and $\mathcal{A}(e_{71}) = 3$. Next, the adjacent-to-vertex map $\mathcal{A}^{-1}$ must be updated so that $\mathcal{A}^{-1}(1)$ now has e_{37} added into it, $\mathcal{A}^{-1}(3)$ now includes e_{71}, and $\mathcal{A}^{-1}(7)$ now includes e_{13}. Next, we update the graph $\mathcal{N}$

Algorithm 7 Testing if a point is to the left of a line.

Inputs:

- A line L_{ij} through p_i and p_j, and a query point p_k.
- A point set $\mathcal{P}$.

Outputs:

- Returns 1 if p_k is to the left of L_{ij}, 0 if p_k is on L_{ij}, and -1 if p_k is to the right of L_{ij}.

function IsLeftOfLine($\mathcal{P}$, i, j, k)
    ($i == -1$ && $j == -3$) && **return** -1
    ($i == -1$ && $j == -2$) && **return** 1
    ($i == -3$ && $j == -1$) && **return** 1
    ($i == -3$ && $j == -2$) && **return** -1
    ($i == -2$ && $j == -3$) && **return** 1
    ($i == -2$ && $j == -1$) && **return** -1
    return `IsOriented`($\mathcal{P}(i)$, $\mathcal{P}(j)$, $\mathcal{P}(k)$)
end function

Algorithm 8 Finding what edge of a triangle a point is on.

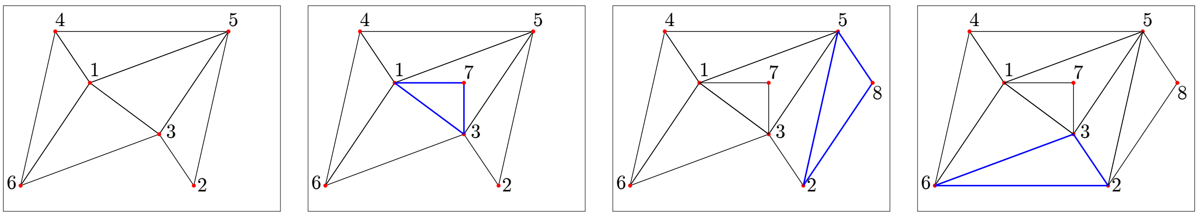
Inputs:

- A point p_r that is on the edge of a triangle T_{ijk}.
- A point set $\mathcal{P}$.

Outputs:

- Returns the edge of T_{ijk} that the point p_r is on.

function FindEdge(T_{ijk}, $\mathcal{P}$, r)
    `IsLeftOfLine`($\mathcal{P}$, i, j, r) $== 0$ && **return** e_{ij}
    `IsLeftOfLine`($\mathcal{P}$, j, k, r) $== 0$ && **return** e_{jk}
    `IsLeftOfLine`($\mathcal{P}$, k, i, r) $== 0$ && **return** e_{ki}
end function



(a) Initial triangulation. (b) Interior addition. (c) Single boundary addition. (d) Double boundary addition.

Figure 1.4: Examples of adding triangles to an existing triangulation. The triangles shown in blue are the new triangles being added.

so that $\mathcal{N}(1)$ includes 3 and 7, $\mathcal{N}(3)$ now includes 1 and 7, and $\mathcal{N}(7)$ now includes 1 and 3. Note that some of these neighbourhood connections may already be included, but our representation of $\mathcal{N}$ as an undirected graph will handle any duplicates by ignoring them. For this simple case, then, the procedure for adding a triangle T_{ijk} is:

`push!`($\mathcal{T}$, T_{ijk})
$\mathcal{A}(e_{ij}) = k$
$\mathcal{A}(e_{jk}) = i$

$\mathcal{A}(e_{ki}) = j$
push!($\mathcal{A}^{-1}(i)$, e_{jk})
push!($\mathcal{A}^{-1}(j)$, e_{ki})
push!($\mathcal{A}^{-1}(k)$, e_{ij})
push!($\mathcal{N}(k)$, i, j) ▷ This also does e.g. push!($\mathcal{N}(i), k$) as $\mathcal{N}$ is undirected.
push!($\mathcal{N}(i)$, j)

Now consider Figure 1.4c where we are adding T_{528}. Firstly, note that we can identify that this triangle is being added onto a boundary edge since $\mathcal{A}(e_{52}) = \partial$. Once we have identified that an edge of the triangle to be added forms part of the boundary, we need to also confirm that the other edges form part of the boundary – if p_8 were inside T_{325}, then e_{58} and e_{82} would not be boundary edges, but e_{52} would still be a boundary edge. Thankfully, since we we only work with positively oriented triangles in all these codes, there is no actual need to check this. Now, with the addition of these triangle, we apply the same procedure as before. In addition, we must set $\mathcal{A}(e_{58}) = \partial$ and $\mathcal{A}(e_{82}) = \partial$, and also $\mathcal{A}^{-1}(\partial)$ must now include e_{58} and e_{82} and exclude e_{52}. So, assuming the single boundary edge is e_{ij}, the code for adding a triangle T_{ijk} that has a single boundary edge is:

push!($\mathcal{T}$, T_{ijk})
$\mathcal{A}(e_{ij}) = k$
$\mathcal{A}(e_{jk}) = i$
$\mathcal{A}(e_{ki}) = j$
push!($\mathcal{A}^{-1}(i)$, e_{jk})
push!($\mathcal{A}^{-1}(j)$, e_{ki})
push!($\mathcal{A}^{-1}(k)$, e_{ij})
push!($\mathcal{N}(k)$, i, j)
push!($\mathcal{N}(i)$, j)
$\mathcal{A}(e_{ik}) = \partial$
$\mathcal{A}(e_{kj}) = \partial$
push!($\mathcal{A}^{-1}(\partial)$, e_{ik}, e_{kj})
delete!($\mathcal{A}^{-1}(\partial)$, e_{ij})

Now let us finally consider the case in Figure 1.4d where we are adding T_{623}. In this case, we once again do the same updates as before. For the boundary updates, noting that $\mathcal{A}(e_{23}) = \mathcal{A}(e_{36}) = \partial$, we set $\mathcal{A}(e_{26}) = \partial$ and remove e_{23} and e_{36} from $\mathcal{A}^{-1}(\partial)$, respectively. Moreover, we now include e_{26} in $\mathcal{A}^{-1}(\partial)$. Notice that while the triangle is T_{623} the relevant boundary edge is e_{26}, reversing the orientation of the edge e_{62} of T_{623}. So, letting e_{jk} and e_{ki} be the previous boundary edges and e_{ij} the new boundary edge, the procedure for adding a triangle T_{ij} that has two boundary edges is:

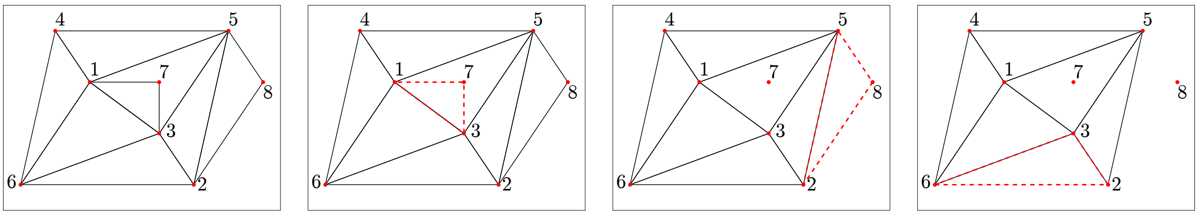
push!($\mathcal{T}$, T_{ijk})
$\mathcal{A}(e_{ij}) = k$
$\mathcal{A}(e_{jk}) = i$
$\mathcal{A}(e_{ki}) = j$
push!($\mathcal{A}^{-1}(i)$, e_{jk})
push!($\mathcal{A}^{-1}(j)$, e_{ki})
push!($\mathcal{A}^{-1}(k)$, e_{ij})
push!($\mathcal{N}(k)$, i, j)
push!($\mathcal{N}(i)$, j)
$\mathcal{A}(e_{ji}) = \partial$

push!($\mathcal{A}^{-1}(\partial)$, e_{ji})
delete!($\mathcal{A}^{-1}(\partial)$, e_{jk}, e_{ki})

There is another special case to consider, which is the addition of a triangle with three boundary edges. In this case, the triangulation we are adding onto must be empty. So, for a triangle T_{ijk}, we just add it as normal but also set $\mathcal{A}(e_{kj}) = \mathcal{A}(e_{ji}) = \mathcal{A}(e_{ik}) = \partial$ and push e_{kj}, e_{ji}, and e_{ik} into $\mathcal{A}^{-1}(\partial)$. Algorithm 9 gives a complete description of our algorithm with this special case considered, with an empty triangulation detected by checking if $|\mathcal{T}| = 1$ after T_{ijk} is added. Note that Algorithm 9 also adds boundary points into $\mathcal{N}(\partial)$.

1.5.3 Deleting a triangle

The next problem is that of deleting a triangle. Ideally, the deletion of T_{ijk} should be the inverse of adding T_{ijk}. With this goal in mind, let us work through the examples in Figure 1.4. To start simple, though, we will follow the order in Figure 1.5.



(a) Initial triangulation. (b) Interior deletion. (c) Double boundary deletion. (d) Single boundary deletion.

Figure 1.5: Examples of deleting triangles to an existing triangulation. The triangles shown in red are the triangles being deleted.

Let us start by discussing Figure 1.5b where we are deleting T_{137}. This case of an interior deletion is simple. First, we must delete the keys e_{13}, e_{37}, and e_{71} from $\mathcal{A}$, Similarly, we remove e_{13}, e_{37}, and e_{71} from $\mathcal{A}^{-1}(7)$, $\mathcal{A}^{-1}(1)$, and $\mathcal{A}^{-1}(3)$, respectively. Next, we delete 1 and 3 from $\mathcal{N}(7)$. For the neighbourhoods of 1 and 3, we do not delete 3 from $\mathcal{N}(1)$ or 1 from $\mathcal{N}(3)$ as we know that the other edge e_{31} does exist in $\mathcal{A}$. So, we have the following procedure for deleting a triangle with no boundary edges:

delete!($\mathcal{T}$, T_{ijk})
delete!($\mathcal{A}$, e_{ij}, e_{jk}, e_{ki})
delete!($\mathcal{A}^{-1}(i)$, e_{jk})
delete!($\mathcal{A}^{-1}(j)$, e_{ki})
delete!($\mathcal{A}^{-1}(k)$, e_{ij})
$v_{ji} = \mathcal{A}(e_{ji}) \neq \emptyset$
$v_{ik} = \mathcal{A}(e_{ik}) \neq \emptyset$
$v_{kj} = \mathcal{A}(e_{kj}) \neq \emptyset$
!v_{ji} && delete!($\mathcal{N}(i)$, j)
!v_{ik} && delete!($\mathcal{N}(k)$, i)
!v_{kj} && delete!($\mathcal{N}(j)$, k)

The next step is to consider deleting the triangle with two boundary edges, as indicated in Figure 1.5c where we are deleting T_{528}. In this case, we can do the same deletions as before. The only new thing to note is that, as e_{58} and e_{82} are both boundary edges,

Algorithm 9 Adding a triangle into an existing triangulation.

Inputs:

- A triangle T_{ijk} to be added into a triangulation $\mathcal{T}$.
- The adjacent map $\mathcal{A}$, the adjacent-to-vertex map $\mathcal{A}^{-1}$, and the graph $\mathcal{N}$.

Outputs:

- An updated triangulation $\mathcal{T}$ that now includes T_{ijk}.

function $\textsc{AddTriangle}(i, j, k, \mathcal{T}, \mathcal{A}, \mathcal{A}^{-1}, \mathcal{N})$
  `push!`$(\mathcal{T}, T_{ijk})$
  $b_{ij} =$ `IsBoundaryEdge`$(i, j, \mathcal{A})$
  $b_{jk} =$ `IsBoundaryEdge`$(j, k, \mathcal{A})$
  $b_{ki} =$ `IsBoundaryEdge`$(k, i, \mathcal{A})$
  $m = b_{ij} + b_{jk} + b_{ki}$ ▷ Number of boundary edges.
  $\mathcal{A}(e_{ij}) = k$
  $\mathcal{A}(e_{jk}) = i$
  $\mathcal{A}(e_{ki}) = j$
  `push!`$(\mathcal{A}^{-1}(i), e_{jk})$
  `push!`$(\mathcal{A}^{-1}(j), e_{ki})$
  `push!`$(\mathcal{A}^{-1}(k), e_{ij})$
  `push!`$(\mathcal{N}(k), i, j)$
  `push!`$(\mathcal{N}(i), j)$
  $m == 1$ `&&` `AddBoundaryEdgesSingle`$(i, j, k, b_{ij}, b_{jk}, b_{ki}, \mathcal{A}, \mathcal{A}^{-1}, \mathcal{N})$
  $m == 2$ `&&` `AddBoundaryEdgesDouble`$(i, j, k, b_{ij}, b_{jk}, b_{ki}, \mathcal{A}, \mathcal{A}^{-1}, \mathcal{N})$
  $|\mathcal{T}| == 1$ `&&` `AddBoundaryEdgesTriple`$(i, j, k, \mathcal{A}, \mathcal{A}^{-1}, \mathcal{N})$
end function
function $\textsc{AddBoundaryEdgesSingle}(i, j, k, b_{ij}, b_{jk}, b_{ki}, \mathcal{A}, \mathcal{A}^{-1}, \mathcal{N})$
  $u, v, w =$ `RotateTriangle`$(b_{ij}, b_{jk}, b_{ki}, i, j, k)$
  $\mathcal{A}(e_{uw}) = \partial$
  $\mathcal{A}(e_{wv}) = \partial$
  `push!`$(\mathcal{A}^{-1}(\partial), e_{uw}, e_{wv})$
  `delete!`$(\mathcal{A}^{-1}(\partial), e_{uv})$
  `push!`$(\mathcal{N}(\partial), w)$
end function
function $\textsc{AddBoundaryEdgesDouble}(i, j, k, b_{ij}, b_{jk}, b_{ki}, \mathcal{A}, \mathcal{A}^{-1}, \mathcal{N})$
  $u, v, w =$ `RotateTriangle`$(!b_{ij}, !b_{jk}, !b_{ki}, i, j, k)$
  $\mathcal{A}(e_{vu}) = \partial$
  `push!`$(\mathcal{A}^{-1}(\partial), e_{vu})$
  `delete!`$(\mathcal{A}^{-1}(\partial), e_{vw}, e_{wu})$
  `delete!`$(\mathcal{N}(\partial), w)$
end function
function $\textsc{AddBoundaryEdgesTriple}(i, j, k, \mathcal{A}, \mathcal{A}^{-1}, \mathcal{N})$
  $\mathcal{A}(e_{ji}) = \partial$
  $\mathcal{A}(e_{ik}) = \partial$
  $\mathcal{A}(e_{kj}) = \partial$
  `push!`$(\mathcal{A}^{-1}(\partial), e_{ji}, e_{ik}, e_{kj})$
  `push!`$(\mathcal{N}(\partial), i, j, k)$
end function

deleting them must leave the other edge as a boundary edge, namely e_{52}. So, we must set $\mathcal{A}(e_{52}) = \partial$ and push e_{52} into $\mathcal{A}^{-1}(\partial)$. Next, we delete $\mathcal{A}(e_{58})$ and $\mathcal{A}(e_{82})$, and remove e_{58} and e_{82} from $\mathcal{A}^{-1}(\partial)$. Note that in the case of a boundary edge, the edge will exist. For example, if e_{ji} is a boundary edge, then v_{ji} will be true in Line 6 above, and so we will not delete j from the neighbourhood of i in Line 9, when really it should be deleted. So, assuming that e_{ik} and e_{kj} are the two boundary edges, so that e_{ij} is the new boundary edge, we obtain the following procedure for deleting a triangle with two boundary edges:

1: `delete!`$(\mathcal{T}, T_{ijk})$
2: `delete!`$(\mathcal{A}, e_{ij}, e_{jk}, e_{ki})$
3: `delete!`$(\mathcal{A}^{-1}(i), e_{jk})$
4: `delete!`$(\mathcal{A}^{-1}(j), e_{ki})$
5: `delete!`$(\mathcal{A}^{-1}(k), e_{ij})$
6: `delete!`$(\mathcal{N}(j), k)$
7: `delete!`$(\mathcal{N}(k), i)$
8: `delete!`$(\mathcal{A}, e_{ik}, e_{kj})$
9: $\mathcal{A}(e_{ij}) = \partial$
10: `push!`$(\mathcal{A}^{-1}(\partial), e_{ij})$
11: `delete!`$(\mathcal{A}^{-1}(\partial), e_{ik}, e_{kj})$

Now we discuss the last case in Figure 1.5d where we are deleting the triangle T_{623} that has a single boundary edge. Here, we again apply the same procedure for deleting a triangle as before, but we need to take care of the single boundary edge e_{26} and how it gets converted into the boundary edges e_{23} and e_{36}. So, we delete $\mathcal{A}(e_{26})$ and delete e_{26} from $\mathcal{A}^{-1}(\partial)$. Next, we set $\mathcal{A}(e_{23}) = \partial$ and $\mathcal{A}(e_{36}) = \partial$ and add e_{23} and e_{36} to $\mathcal{A}^{-1}(\partial)$. So, assuming that e_{ik} is the current boundary edge so that the new boundary edges to be added are e_{ij} and e_{jk}:

1: `delete!`$(\mathcal{T}, T_{ijk})$
2: `delete!`$(\mathcal{A}, e_{ij}, e_{jk}, e_{ki})$
3: `delete!`$(\mathcal{A}^{-1}(i), e_{jk})$
4: `delete!`$(\mathcal{A}^{-1}(j), e_{ki})$
5: `delete!`$(\mathcal{A}^{-1}(k), e_{ij})$
6: `delete!`$(\mathcal{N}(k), i)$
7: `delete!`$(\mathcal{A}, e_{ik})$
8: $\mathcal{A}(e_{ij}) = \partial$
9: $\mathcal{A}(e_{jk}) = \partial$
10: `push!`$(\mathcal{A}^{-1}(\partial), e_{ij}, e_{jk})$
11: `delete!`$(\mathcal{A}^{-1}(\partial), e_{ik})$

We note that there is also the case of deleting a triangle with three boundary edges, in which the case the entire triangulation is that triangle. Keeping this special case in mind, we obtain Algorithm 10. Note that `IsBoundaryEdge` and `RotateTriangle` in Algorithm 10 were defined in Algorithm 9. The `protect_boundary` keyword in Algorithm 10 is in case we do not want to delete any boundary edges, for example when splitting a triangle as discussed in the next section. Note that Algorithm 10 also adds boundary points into $\mathcal{N}(\partial)$.

Algorithm 10 Deleting a triangle from an existing triangulation.

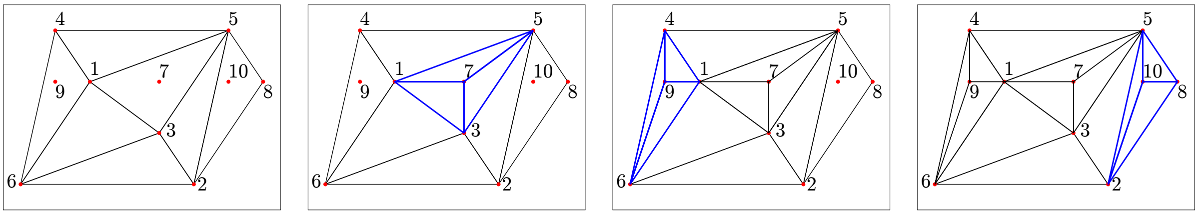
Inputs:

- A triangle T_{ijk} to be added into a triangulation $\mathcal{T}$ and the adjacent map $\mathcal{A}$, the adjacent-to-vertex map $\mathcal{A}^{-1}$, and the graph $\mathcal{N}$.

Outputs:

- An updated triangulation $\mathcal{T}$ that now excludes T_{ijk}.

function DeleteTriangle(i, j, k, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$; `protect_boundary = false`)
    `delete!`($\mathcal{T}$, T_{ijk})
    `delete!`($\mathcal{A}$, e_{ij}, e_{jk}, e_{ki})
    `delete!`($\mathcal{A}^{-1}(i)$, e_{jk})
    `delete!`($\mathcal{A}^{-1}(j)$, e_{ki})
    `delete!`($\mathcal{A}^{-1}(k)$, e_{ij})
    b_{ji} = `IsBoundaryEdge`(j, i, $\mathcal{A}$)
    b_{ik} = `IsBoundaryEdge`(i, k, $\mathcal{A}$)
    b_{kj} = `IsBoundaryEdge`(k, j, $\mathcal{A}$)
    m = `!protect_boundary ?` $b_{ji} + b_{ik} + b_{kj}$ `:` 0 ▷ Number of boundary edges.
    for $e_{rs} \in \{e_{ji}, e_{ik}, e_{kj}\}$ **do**
        v_{rs} = `EdgeExists`(r, s, $\mathcal{A}$)
        $p_{rs} = v_{rs}$ `&& !`b_{rs} ▷ Only protect e_{rs} if it exists and is not a boundary edge.
        `!`p_{rs} `&& delete!`($\mathcal{N}(r)$, s)
    end for
    $m == 1$ `&& DeleteBoundaryEdgesSingle`(i, j, k, b_{ji}, b_{ik}, b_{kj}, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
    $m == 2$ `&& DeleteBoundaryEdgesDouble`(i, j, k, b_{ji}, b_{ik}, b_{kj}, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
    $m == 3$ `&& DeleteBoundaryEdgesTriple`(i, j, k, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
end function
function DeleteBoundaryEdgesSingle(i, j, k, b_{ji}, b_{ik}, b_{kj}, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
    u, v, w = `RotateTriangle`(b_{ji}, b_{kj}, b_{ik}, i, j, k)
    `delete!`($\mathcal{A}$, e_{vu})
    `delete!`($\mathcal{A}^{-1}(\partial)$, e_{vu})
    $\mathcal{A}(e_{vw}) = \partial$
    $\mathcal{A}(e_{wu}) = \partial$
    `push!`($\mathcal{A}^{-1}(\partial)$, e_{vw}, e_{wu})
    `push!`($\mathcal{N}(\partial)$, w)
end function
function DeleteBoundaryEdgesDouble(i, j, k, b_{ji}, b_{ik}, b_{kj}, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
    u, v, w = `RotateTriangle`(b_{ji}, b_{kj}, b_{ik}, i, j, k)
    `delete!`($\mathcal{A}$, e_{uw}, e_{wv})
    `delete!`($\mathcal{A}^{-1}(\partial)$, e_{uw}, e_{wv})
    $\mathcal{A}(e_{uv}) = \partial$
    `push!`($\mathcal{A}^{-1}(\partial)$, e_{uv})
    `delete!`($\mathcal{N}(\partial)$, w)
end function
function DeleteBoundaryEdgesTriple(i, j, k, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
    `delete!`($\mathcal{A}$, e_{kj}, e_{ji}, e_{ik})
    `delete!`($\mathcal{A}^{-1}(\partial)$, e_{kj}, e_{ji}, e_{ik})
    `delete!`($\mathcal{N}(\partial)$, i, j, k)
end function



(a) Initial triangulation. (b) Splitting of T_{135} about p_7. (c) Splitting of T_{614} about p_9. (d) Splitting of T_{528} about p_{10}.

Figure 1.6: Examples of splitting triangles in the interior in an existing triangulation. The triangles shown in blue are new triangles added after splitting.

1.5.4 Splitting a triangle in the interior

Now we consider the problem of splitting a triangle in the interior. This means taking some triangle T_{ijk} and a point p_r in its interior. We then subdivide T_{ijk} into the three triangles T_{ijr}, T_{jkr}, and T_{kir}. Figure 1.6 shows some examples of these subdivisions. Provided our algorithms for deleting a triangle and adding a triangle are working correctly, then this should be as simple as deleting the triangle T_{ijk} and then adding the triangles T_{ijr}, T_{jkr}, and T_{kir}. There is one problem, though. Consider Figure 1.6c. If we delete T_{614} and then add T_{469} first, then there is a hole defined by the vertices (p_4, p_9, p_6, p_1) which causes issues later when updating the boundary accordingly. To remedy this, we introduce into Algorithm 10 a keyword `protect_boundary` that will allow for the boundary edges to be protected, noting that splitting a triangle into three will never change the boundary. Keeping this in mind, we obtain Algorithm 11 for splitting a triangle.

Algorithm 11 Splitting a triangle in the interior.

Inputs:

- A triangle T_{ijk} and a point p_r in T_{ijk}'s interior that will be used to subdivide T_{ijk} into three triangles.
- An existing triangulation $\mathcal{T}$, the adjacent map $\mathcal{A}$, the adjacent-to-vertex map $\mathcal{A}^{-1}$, and the graph $\mathcal{N}$.

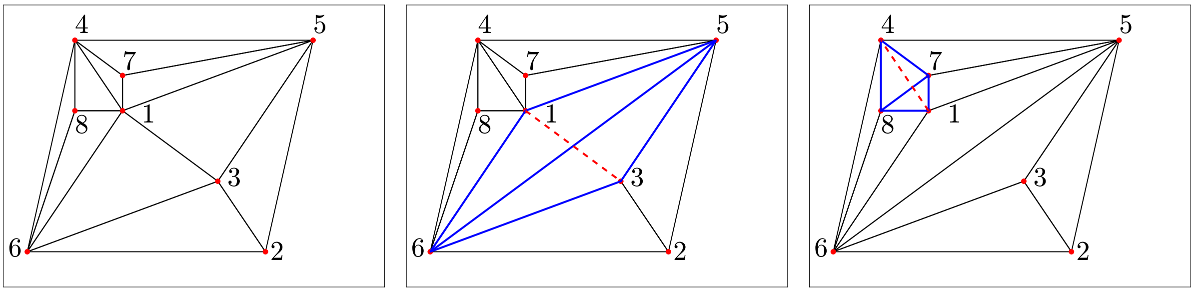
Outputs:

- An updated triangulation $\mathcal{T}$ that has now split T_{ijk} into the three triangles T_{ijr}, T_{jkr}, T_{kir}.

function SplitTriangle(i, j, k, r, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
  `DeleteTriangle`(i, j, k, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$; `protect_boundary = true`)
  `AddTriangle`(i, j, r, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
  `AddTriangle`(j, k, r, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
  `AddTriangle`(k, i, r, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
end function

1.5.5 Flipping an edge

An important operation to perform on a triangulation is that of flipping an edge, with the aim of making it a legal edge (as discussed in the next section). If we have an edge e_{ij} that is incident to the triangles T_{ikj} and $T_{ij\ell}$, then an edge flip of e_{ij} means replacing the edge e_{ij} (and e_{ji}) with $e_{k\ell}$ (and $e_{\ell k}$), which also means replacing T_{ikj} and $T_{ij\ell}$ with



(a) Initial triangulation. (b) The edge e_{13} was flipped to e_{65}. (c) The edge e_{41} was flipped to e_{87}.

Figure 1.7: Examples of edge flipping. In the last two figures, the blue triangles are the new triangles and the red dashed line shows the position of the original edge prior to flipping.

$T_{\ell kj}$ and $T_{\ell ik}$. Examples of some edge flips are shown in Figure 1.7. Note that this flip only makes sense if the quadrilateral defined by (p_i, p_k, j, ℓ) is convex, else the new edge will cross another and force the triangulation to no longer be planar. An edge flip can be implemented with two `DeleteTriangles` and `AddTriangles`, making sure we use the `protect_boundary` keyword in Algorithm 10 as in Algorithm 11, noting that we will never flip a boundary edge. The algorithm that we end up with is given in Algorithm 12.

Algorithm 12 Flipping an edge.

Inputs:

- An edge e_{ij} to be flipped, assuming (p_i, p_k, p_j, p_ℓ) is a convex quadrilateral, where $\ell = \mathcal{A}(e_{ij})$ and $k = \mathcal{A}(e_{ji})$.
- An existing triangulation $\mathcal{T}$, the adjacent map $\mathcal{A}$, the adjacent-to-vertex map $\mathcal{A}^{-1}$, and the graph $\mathcal{N}$.

Outputs:

- An updated triangulation $\mathcal{T}$ that has now flipped e_{ij}.

function FLIPEDGE(i, j, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
    $\ell = \mathcal{A}(e_{ij})$
    $k = \mathcal{A}(e_{ji})$
    `DeleteTriangle`(i, k, j, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$; `protect_boundary = true`)
    `DeleteTriangle`(i, j, ℓ, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$; `protect_boundary = true`)
    `AddTriangle`(ℓ, k, j, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
    `AddTriangle`(ℓ, i, k, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
end function

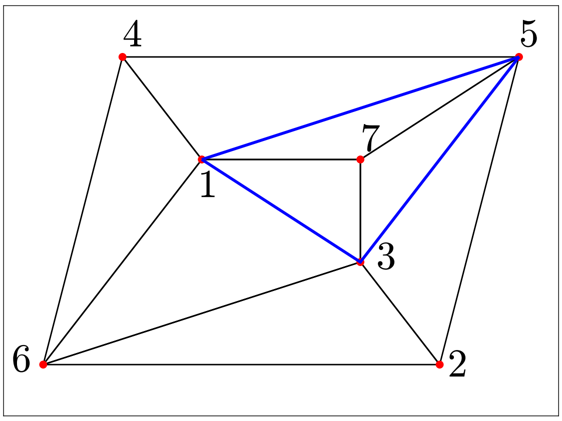
1.5.6 Legalising an edge

When we split a triangle into three, edges in the triangulation may no longer be Delaunay, i.e. no longer legal. Suppose we have split some triangle T_{ijk} into three around a point p_r, subdividing T_{ijk} into T_{ijr}, T_{jkr}, and T_{kir}. It is not difficult to see that the edges e_{ri}, e_{rj}, and e_{rk} are all legal. To see this, note that T_{ijk} is Delaunay prior to the addition of p_r. Therefore, the open circumcircle C of T_{ijk} contains no other points in its interior. We can shrink C to find a circle C' touching both i and r, and since $C' \subset C$ we see that C'

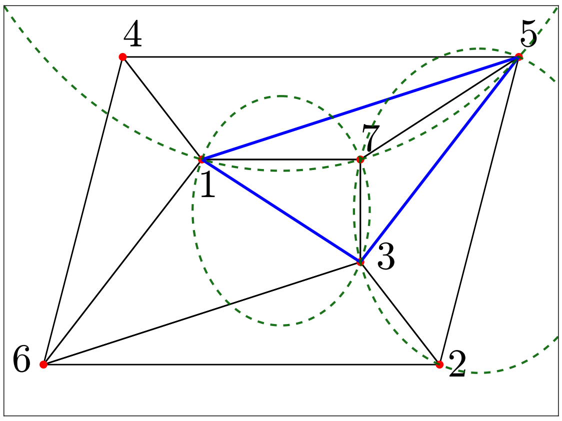
will also contain no points in its interior. In particular, after the addition of r, the edge e_{ri} will be legal. We can apply the same ideas to the edges e_{rj} and e_{rk}. Thus, all the new edges that are introduced upon splitting a triangle are legal.

To understand what edges could become illegal after adding p_r, we need to understand what can cause a previous legal edge e_{ij} to become illegal. Let T_{ijk} and $T_{ji\ell}$ be the edges that e_{ij} is incident to (if e_{ij} is a boundary edge so that there is only one incident triangle, it is legal as it forms part of the convex hull – unless p_r is added outside of the domain; we consider this later). The only way for e_{ij} to be illegal is if one of T_{ijk} and $T_{ji\ell}$ has changed. To understand why it cannot be any other triangle, see that if the modification of some other triangle were to somehow affect whether or not e_{ij} is legal, this would imply that its open circumdisk touches both p_i and p_jj. But, if this triangle is not T_{ijk} or $T_{ji\ell}$, this circumdisk would have to contain either p_k or p_ℓ, meaning these triangles are not Delaunay. Thus, we need only consider the triangles incident to an edge to detect whether it is now illegal.

Now let us bring this back to the problem of legalising an edge. The above discussion tells us that, in the case of the new triangle in Figure 1.8, we need only consider legalising the edges e_{13}, e_{35}, and e_{51}. These edges are highlighted in Figure 1.8a. The check for whether the edges are illegal relies on Lemma 1.1. In particular, we can check if the edge e_{ij}, incident to triangles T_{ijk} and $T_{ji\ell}$, is illegal by checking if p_ℓ is inside the circle touching p_i, p_j, and p_k, using a predicate `IsInCircle` to do this check. This predicate was defined in Algorithm 4. If p_ℓ is inside this circle, the edge is illegal. In the case of Figure 1.8a, letting C_{ijk} be the circle through p_i, p_j, and p_k, we need to check if p_6 is inside the circle C_{137} for e_{13}, if p_2 is inside the circle C_{357} for e_{35}, and if p_4 is inside the circle C_{517} for e_{51}. We show these circles in Figure 1.8b. We see in Figure 1.8b that the edge e_{13} is legal as p_6 is not inside C_{137}; the edge e_{35} is legal since p_2 is not in the interior of C_{357}, although p_2 is circular with p_3, p_5, and p_7, which is not a problem; the edge e_{51} is illegal as p_4 is inside C_{517}.



(a) The addition of the point 7 splits the triangle T_{135} into T_{137}, T_{357}, and T_{175}. The blue edges shown need to be checked in case they are now illegal.



(b) The circles C_{137}, C_{357}, and C_{517} used to test whether the edges $(1, 3)$, $(3, 5)$, and $(5, 1)$ are illegal, respectively.

Figure 1.8: Legalising a triangle after splitting.

We now need to legalise the edge e_{51}. This is done by flipping the edge e_{51} to become e_{41}, using Algorithm 12. Once we have flipped this edge, we know that we will have the new triangles T_{417} and T_{547}, deleting T_{175} and T_{415}. We therefore need to ensure that the edges e_{54} and e_{41} are still legal, so we apply the same flipping process. This leads to the recursive algorithm in Algorithm 13 for legalising an edge through edge flipping.

Algorithm 13 Legalising an edge.

Inputs:

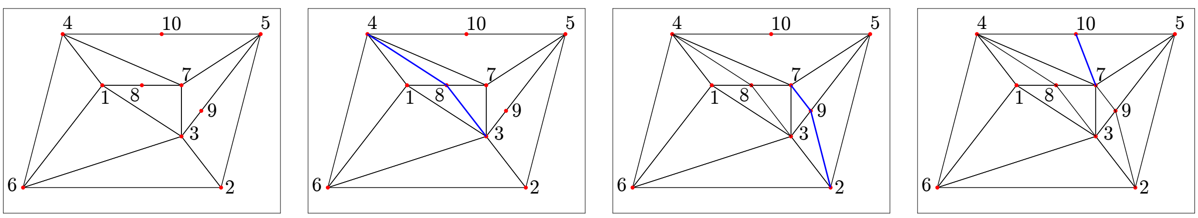
- An edge e_{ij} to legalise after a point p_r was added into T_{ijk} following an application of Algorithm 11.
- An existing triangulation $\mathcal{T}$, the adjacent map $\mathcal{A}$, the adjacent-to-vertex map $\mathcal{A}^{-1}$, the graph $\mathcal{N}$, and the point set $\mathcal{P}$.

Outputs:

- An updated triangulation $\mathcal{T}$ such that e_{ij} is now legal.

procedure LegaliseEdge(i, j, r, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$, $\mathcal{P}$)
    if `IsLegal`(i, j, $\mathcal{A}$, $\mathcal{P}$) **then**
        $\ell = \mathcal{A}(e_{ji})$
        `FlipEdge`(i, j, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
        `LegaliseEdge`(i, ℓ, r, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$, $\mathcal{P}$)
        `LegaliseEdge`(ℓ, j, r, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$, $\mathcal{P}$)
    end if
end procedure
procedure IsLegal(i, j, $\mathcal{A}$, $\mathcal{P}$)
    $k = \mathcal{A}(e_{ij})$
    $\ell = \mathcal{A}(e_{ji})$
    $e =$ `IsInCircle`($\mathcal{P}$, i, j, k, ℓ)
    return $e \geq 0$
end procedure

1.5.7 Splitting an edge



(a) Original triangulation. (b) Splitting the edge e_{17} at p_8. (c) Splitting the edge e_{35} at p_9. (d) Splitting the edge e_{45} at p_{10}.

Figure 1.9: Examples of splitting an edge. In each figure, the new edges are shown in blue.

When we add points into an existing triangulation, issues may arise if the point to be added is on an edge of another triangle. One algorithm we implement adds points and then splits triangles in the interior using Algorithm 11, but this fails in this case. When a point is on the edge, we instead make new triangles by drawing edges to the adjacent vertices. Examples of this splitting are shown in Figure 1.9, which shows the two different cases. In Figure 1.9b we are splitting an interior edge e_{17} at a point p_8, and this does not introduce any new boundary edges. Similarly, Figure 1.9c shows the splitting of the interior edge e_{35} at a point p_9. The next case in Figure 1.9d shows the splitting of the boundary edge e_{54} at a point p_{10}, and this introduces the new boundary edges $e_{4,10}$ and $e_{10,5}$, deleting the previous boundary edge e_{45}. Notice in this splitting that there are two views that we could take. The first view is that the edge e_{ij} is split in both directions, for example the splitting of e_{17} at p_8 in Figure 1.9b connects p_8 to both p_3 and p_4. An alternative view is that the splitting only goes to the adjacent vertex, so that e_{17} in Figure 1.9b would only be split so that p_8 connects to p_4, and we would have to split e_{71} to get the connection with p_3. This latter view will be the simplest to work with, both for debugging and for dealing with boundary edges, and so this is the view that we take.

Let us start by discussing the splitting of an interior edge, using Figure 1.9b as an example. Here, p_{17} is split at p_8, and this introduces the new triangles T_{184}, T_{874}, T_{138}, and T_{837}, deleting the previous triangles T_{174} and T_{137} in the process. As mentioned, though, we will only consider the new connection with the adjacent vertex p_4. We can therefore represent this splitting using one `DeleteTriangles` and two `AddTriangles`, using the following procedure for splitting an interior edge e_{ij} about a point p_r:

1: $k = \mathcal{A}(e_{ij})$.
2: `DeleteTriangle`$(i,\ j,\ k,\ \mathcal{T},\ \mathcal{A},\ \mathcal{A}^{-1},\ \mathcal{N}$; `protect_boundary = true`$)$
3: `AddTriangle`$(i,\ r,\ k,\ \mathcal{T},\ \mathcal{A},\ \mathcal{A}^{-1},\ \mathcal{N})$
4: `AddTriangle`$(r,\ j,\ k,\ \mathcal{T},\ \mathcal{A},\ \mathcal{A}^{-1},\ \mathcal{N})$

The `protect_boundary = true` case is needed when the edge is part of a boundary triangle, as in Figure 1.9c, since the order in which we add triangles may introduce holes in the domain temporarily; this is the same issue we had when developing Algorithm 11. This procedure could be called on e_{ji} to get the splitting in the other direction.

Now let us consider the case of a boundary edge as in Figure 1.9d. Here, the edge e_{54} is being split at p_{10}, deleting the triangle T_{547} and introducing the two new triangles $T_{7,5,10}$ and $T_{10,4,7}$. We can apply the same procedure as above, but we have to be careful with the boundary edge, ensuring we delete e_{45} and instead replace it with $e_{4,10}$ and $e_{10,5}$. This is as simple as changing the `protect_boundary` argument above to depend on the boundary status of e_{ji}:

1: $k = \mathcal{A}(e_{ij})$.
2: `DeleteTriangle`$(i,\ j,\ k,\ \mathcal{T},\ \mathcal{A},\ \mathcal{A}^{-1},\ \mathcal{N}$; `protect_boundary = !IsBoundaryEdge`$(j,\ i,\ \mathcal{A}))$
3: `AddTriangle`$(i,\ r,\ k,\ \mathcal{T},\ \mathcal{A},\ \mathcal{A}^{-1},\ \mathcal{N})$
4: `AddTriangle`$(r,\ j,\ k,\ \mathcal{T},\ \mathcal{A},\ \mathcal{A}^{-1},\ \mathcal{N})$

Remember that `IsBoundaryEdge` is defined in Algorithm 1. To summarise, Algorithm 14 gives the algorithm for splitting an edge.

1.5.8 Point location by walking

Point location is a common problem in computational geometry, and is an essential feature of many Delaunay triangulation algorithms. While we will discuss many methods for point location, we delay the discussion of these methods until we discuss the specific algorithms. Here we discuss a method that works independent of a specific algorithm. The problem is this: *Given a Delaunay triangulation* $\mathcal{DT}(\mathcal{P})$ *of a point set* $\mathcal{P}$*, and a point* $q \in \mathbb{R}^2$*, what triangle* $T \in \mathcal{DT}(\mathcal{P})$ *contains the point* q*?* The solution we present to

Algorithm 14 Splitting an edge.

Inputs:
- An edge e_{ij} to split at a point p_r.
- An existing triangulation $\mathcal{T}$, the adjacent map $\mathcal{A}$, the adjacent-to-vertex map $\mathcal{A}^{-1}$, and the graph $\mathcal{N}$.

Outputs:
- An updated triangulation $\mathcal{T}$ such that e_{ij} is now split at p_r.

function SPLITEDGE(i, j, r, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
    $k = \mathcal{A}(e_{ij})$.
    `DeleteTriangle`(i, j, k, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$; `protect_boundary = !IsBoundaryEdge`(j, i, $\mathcal{A}$))
    `AddTriangle`(i, r, k, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
    `AddTriangle`(r, j, k, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
end function

this is the jump-and-march algorithm of Mücke et al. (1999), a specific case of what are known as *walking algorithms* like those discussed by Devillers et al. (2002). We discuss the algorithm by Mücke et al. (1999) as it requires minimal setup and is conceptually simple, and is already in common use for example in the Triangle software (Shewchuk, 1996). An orthogonal walk may be of interest as we use exact arithmetic for our geometric predicates, as recommended by Devillers et al. (2002), but we do not consider it here.

The essence of the jump-and-march algorithm is as follows: If $\mathcal{P} = \{p_1, \ldots, p_n\}$ and the query point is q, then:

1. Select m points $(s_1, \ldots, s_m)$ at random and without replacement from $\mathcal{P}$, where $m = \mathcal{O}(n^{1/3})$.

2. Set $j = \operatorname{argmin}_{j=1}^{m} d(s_j, q)$, where $d(x, y)$ is the Euclidean distance between points x and y, and set $p = s_j$.

3. Locate the triangle $T \in \mathcal{DT}(\mathcal{P})$ containing q by traversing all triangles intersected by the line segment $\overline{pq}$.

The runtime of this algorithm for randomly distributed points is $\mathcal{O}(n^{1/3})$ (Devroye et al., 1998). We simply use $m = \lceil n^{1/3} \rceil$, although a choice like $m = \lceil 0.45n^{1/3} \rceil$ may be reasonable for uniformly distributed point sets (Shewchuk, 1996). Algorithm 15 gives an implementation of the first two steps of this procedure. In Algorithm 15 it is important to note that we are actually sampling with replacement, but provided m is much smaller than n this will not be too impactful on the algorithm. In particular, if there are n total points to choose from and we select m points from them, the probability that all the points are unique is

$$\frac{n(n-1)\cdots(n-m+1)}{n^m} = \frac{\Gamma(n+1)}{n^m\Gamma(n-m+1)} \sim 1 - \frac{1}{2n^{1/3}} + \mathcal{O}\left(\frac{1}{n^{2/3}}\right) \quad \text{as } n \to \infty,$$

where $m = n^{1/3}$. This approximation $1 - 1/2n^{1/3}$ appears to have around 10% relative error to the true value for $n \approx 10$, and around 2.8% near $n = 100$. So, the probability of there being any duplicates is approximately $1/2n^{1/3}$, so the tradeoff in avoiding allocations from having to check for duplicates is worth it for large enough n.

Algorithm 15 Selecting an initial point for the jump-and-march algorithm.

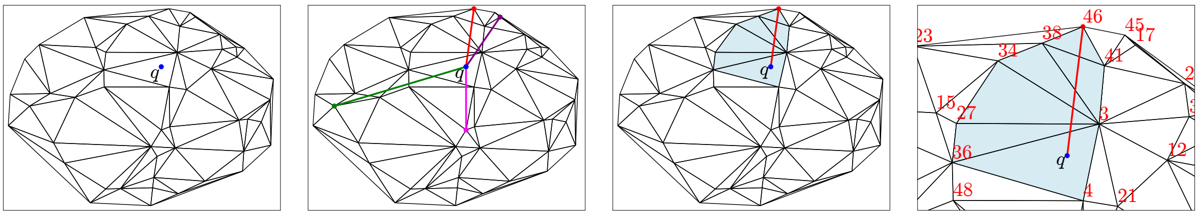
Inputs:

- A query point q, a point set $\mathcal{P}$, and a number of points m to sample.

Outputs:

- A vertex k to start marching from.

function $\textsc{SelectInitialPoint}(\mathcal{P},\ q;\ m = \lceil m^{1/3} \rceil)$
  $\delta^2 = \infty$ ▷ Initialise distance at infinity.
  $k = 0$ ▷ Initialise index for the loop.
  $n = |\mathcal{P}|$
  for $j \in \{1, \ldots, m\}$ **do**
    $i = \texttt{rand}(\{1, \ldots, n\})$
    $p_i = \mathcal{P}(i)$
    $d^2 = (p_{i,x} - q_x)^2 + (p_{i,y} - q_y)^2$
    if $d^2 < \delta^2$ **then**
      $\delta^2 = d^2$
      $k = i$
    end if
  end for
  return k
end function



(a) An initial triangulation and a query point q.

(b) Randomly selected points s_i and line segments $\overrightarrow{s_iq}$.

(c) The triangles stepped over are shown in blue.

(d) Zoomed in and annotated version of (c).

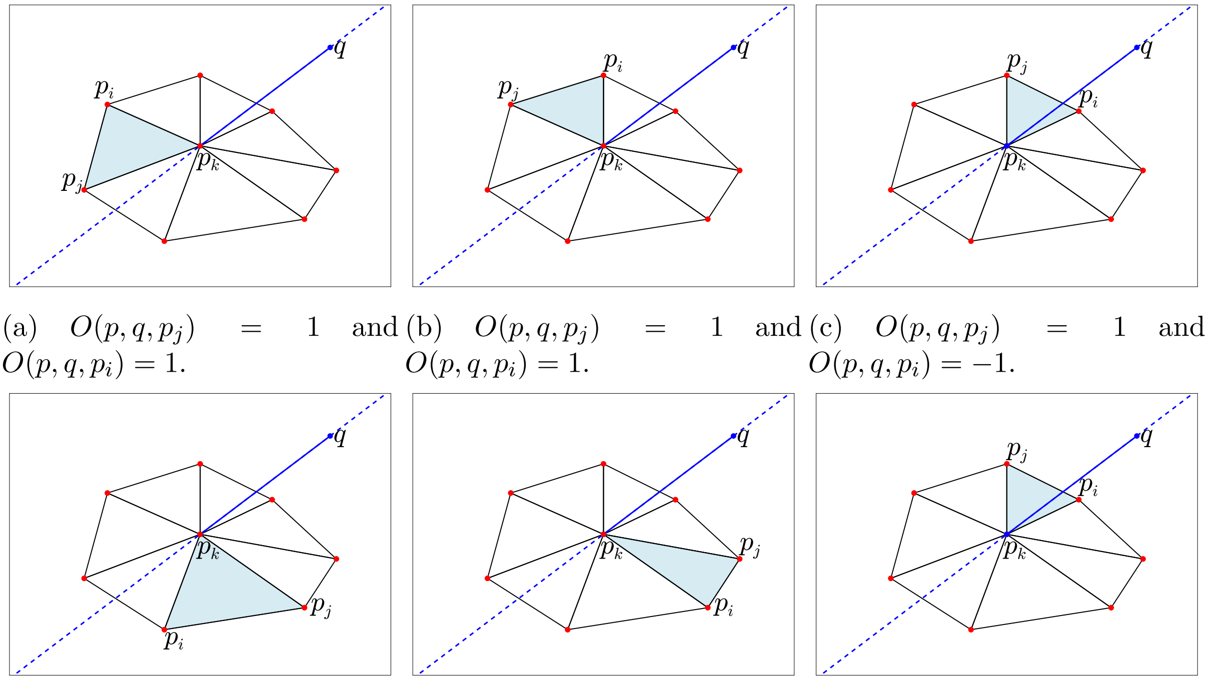
Figure 1.10: An example of point location by walking.

To give an example of this method of point location, Figure 1.10a shows an initial triangulation and some point q inside the triangulation. This triangulation has $n = 50$ points, so we select $m = \lceil n^{1/3} \rceil = 4$ vertices at random, giving four line segments to test as shown in Figure 1.10b. The red line segment has the smallest length of the four, and so we start the point location at the red vertex. We then step over triangles starting from this red vertex until we reach q, stepping over all the triangles shown in blue in Figure 1.10c.

To discuss the implementation of this algorithm, let us consider Figure 1.10d. The walk starts at p_{46}, but to determine where to go we need to first find the triangle to start in. To find this triangle, first remember that $\mathcal{A}^{-1}(46)$ will give us all edges e_{uv} such that $T_{u,v,46}$ is a positively oriented triangle, so we can loop over the elements of $\mathcal{A}^{-1}(46)$. Suppose the first triangle we try is $T_{23,38,46}$. Since the edge $e_{23,28}$ does not intersect $\overrightarrow{pq}$, where $s = p_{46}$, we need to rotate to another triangle. The idea is to rotate around until the orientations become opposite, so the next edge we try is $e_{38,41}$. In this case we find that the line does indeed intersect this edge, so we can start the straight line search.

We traverse the triangles, keeping the edge's endpoints on each side of the line, until we eventually find an edge that switches the orientation of the edge relative to q.

The ideas above can be formalised with the following algorithm, following the implementation of Devillers et al. (2002). Algorithm 16 shows the procedure for finding the triangle containing q after we have already selected a vertex k (thus $p = p_k$) to start from, using say Algorithm 15. Lines 2–5 select the initial triangle by randomly selecting an edge from $\mathcal{A}^{-1}(k)$. The way we step across these neighbouring triangles depends on the orientation of p_j relative to $\overrightarrow{pq}$. Figure 1.11 shows how this is done. If we have selected an edge e_{ij}, so that T_{ijk} is positively oriented where $p = p_k$, and if p_j is to the left of $\overrightarrow{pq}$, then we will rotate around the neighbouring triangles clockwise until p_i is now to the right of $\overrightarrow{pq}$, at which point $\overrightarrow{pq}$ must intersect the new e_{ij}. These steps are shown in Figure 1.11a–1.11c. Similarly, if p_j is to the right, then we will need to rotate counter-clockwise until p_j is to the left of $\overrightarrow{pq}$, as shown in Figure 1.11d–1.11f. This initialisation step is executed in Lines 6–22 of Algorithm 16. Lines 11 and 19 are needed in the rare case that we rotate onto a boundary edge, and so we simply restart the algorithm at the other vertex, and similarly for Line 27.



(a) $O(p, q, p_j) = 1$ and $O(p, q, p_i) = 1$.
(b) $O(p, q, p_j) = 1$ and $O(p, q, p_i) = 1$.
(c) $O(p, q, p_j) = 1$ and $O(p, q, p_i) = -1$.

(d) $O(p, q, p_i) = -1$ and $O(p, q, p_j) = -1$.
(e) $O(p, q, p_i) = -1$ and $O(p, q, p_j) = -1$.
(f) $O(p, q, p_i) = -1$ and $O(p, q, p_j) = 1$.

Figure 1.11: Example of finding the initial triangle for stepping towards q from $p_k = p$. The top row shows the case where we search clockwise as p_j is left of $\overrightarrow{pq}$, and the bottom row shows the case where we search counterclockwise as p_j is right of $\overrightarrow{pq}$. The thick blue line is the line segment $\overrightarrow{pq}$ and the dashed line is the line through s and q. In each subcaption, $O(a, b, c)$ is an abbreviation of `IsOriented`$(a,\ b,\ c)$.

Once the initialisation step is complete and we have an edge e_{ij} that $\overrightarrow{pq}$ intersects, the straight walk part of the algorithm begins. In Lines 23 and 24, we swap i and j so that p_i is to the left of $\overrightarrow{pq}$ and p_j is to the left of $\overrightarrow{pq}$. Lines 26–35 are what determines the next edge to cross from e_{ij}. We first find the triangle T_{ijk} that has e_{ij} as its edge by getting $k = \mathcal{A}(e_{ij})$. If this point p_k is to the right of $\overrightarrow{pq}$, then we need to swap j and k

Algorithm 16 Point location with the jump-and-march algorithm.

Inputs:

- A vertex k to start the walk at and a query point q. See also Algorithm 15 for choosing this vertex k randomly.
- An adjacent map $\mathcal{A}$, adjacent-to-vertex map $\mathcal{A}^{-1}$, and point set $\mathcal{P}$.

Outputs:

- A triangle T_{ijk} that contains q in its interior.

```
function JumpAndMarch(k, q, A, A^{-1}, P)
    p = P(k)
    e_{ij} = rand(A^{-1}(k))                        ▷ A random triangle neighbouring p.
    p_i = P(i)
    p_j = P(j)
    if IsOriented(p, q, p_j) == 1 then
        while IsOriented(p, q, p_i) == 1 do
            j = i
            p_j = p_i
            i = A(e_{ik})
            i == ∂ && return JumpAndMarch(j, q, A, A^{-1}, P)
            p_i = P(i)
        end while
    else                                            ▷ IsOriented(p, q, p_j) ≤ 0.
        while IsOriented(p, q, p_j) == −1 do
            i = j
            p_i = p_j
            j = A(e_{kj})
            j == ∂ && return JumpAndMarch(i, q, A, A^{-1}, P)
            p_j = P(j)
        end while
    end if
    i, j = j, i                                     ▷ p_i is left of \vec{pq}, p_j is right of \vec{pq}.
    p_i, p_j = p_j, p_i
    while IsOriented(p_i, p_j, q) == 1 do
        k = A(e_{ij})
        k == ∂ && return JumpAndMarch(i, q, A, A^{-1}, P)
        p_k = P(k)
        if IsOriented(p, q, p_k) == −1 then
            j = k
            p_j = p_k
        else
            i = k
            p_i = p_k
        end if
    end while
    k = A(e_{ji})                                   ▷ Positively oriented triangle.
    return T_{jik}
end function
```

since we know that p_j is to the right of $\overrightarrow{pq}$, and this is checked in Lines 29–31. Similarly, Lines 33 and 34 handle the case where p_k is to the left of $\overrightarrow{pq}$. The core part of this loop is the termination condition in Line 25, which says that we stop traversing the edges once q is to the left of the edge e_{ij}. To understand this, consider for example Figure 1.11c. In this setting, q is to the right of e_{ij}, so we need to keep searching. As soon as e_{ij} is moved to the right of q, though, we will have passed the triangle that q is in, meaning we have arrived at it. Hence, we stop the loop once q is to the left of e_{ij}. The final line in Line 36 just updates the found triangle so that it is positively oriented.

1.5.9 Triangulating convex polygons

We start by discussing a method for triangulating convex polygons. This will be useful later when we discuss the problem of deleting triangles. We follow the discuss of Cheng et al. (2013) who give an algorithm for the original algorithm by Chew (1990).

1.6 Algorithms for Computing the Delaunay Triangulation

Now let us give some algorithms for computing the Delaunay triangulation. We give two separate methods.

1.6.1 de Berg's randomised incremental insertion algorithm

The first algorithm we consider is the algorithm described by de Berg et al. (1999). The algorithm is a randomised incremental insertion method, where we start with an initial triangle and then insert new points in a random order. When a new point is added, `SplitTriangle` (or `SplitEdge`, if the new point is on the edge of an existing triangle) is used to define new triangles, and then `LegaliseEdge` is used to make all the edges legal, thus making the triangulation Delaunay. This is done until all the points have been sorted.

Super triangle

The first issue to deal with is the initial triangle. Following de Berg et al. (1999), we surround the points in a *super triangle*, a triangle that contains all the points in the point set $\mathcal{P}$. While we could handle this triangle's vertices symbolically, for simplicity we will actually define specific coordinates for this super triangle. Let the super triangle be $T_{-1,-2,-3}$ with vertices at p_{-1}, p_{-2}, and p_{-3}. If we have n points, $p_1, p_2, \ldots, p_n$, to be added, each with coordinates $p_i = (x_i, y_i)$, then we define

$$x^m = \min_{i=1}^n x_i, \quad x^M = \max_{i=1}^n x_i, \quad y^m = \min_{i=1}^n y_i, \quad y^M = \max_{i=1}^n y_i.$$

With these definitions, we define a bounding box $[x^m, x^M] \times [y^m, y^M]$ for the point set. The centroid of this bounding box is at $(x^c, y^c) = [(x^m + x^M)/2, (y^m + y^M)/2]$. If we define $\delta = \max\{x^M - x^m, y^m - y^M\}$, then we define

$$p_{-1} = (x^c + M\delta, y^c - \delta), \quad p_{-2} = (x^c, y^c + M\delta), \quad p_{-3} = (x^c - M\delta, y^c - \delta), \tag{1.1}$$

where $M = 27.39$; this value M is just a shift that pushes the super triangle further from the point set. These coordinates will be large enough so that, when they are removed, the Delaunay triangulation of the original point set is obtained.

Point location

The second issue is that of point location: when we add a new point p_r, we need to know what triangle it is on (or what edge it is on) in order to know what triangle we need to split. This can be done by using a *history graph*, which is a directed acyclic graph (DAG) that tracks how the triangles in the algorithm have been split and what triangles they became. To understand how this might work, consider Figure 1.6. In Figure 1.6b, we have split the triangle T_{135} into the three triangles T_{137}, T_{357}, and T_{517}. Our DAG would thus have the nodes T_{135}, T_{137}, T_{357}, and T_{517}, and the node T_{137} would have the out-neighbours T_{137}, T_{357}, and T_{517}. This is useful since if we know that a point is inside T_{137}, and since T_{137}, T_{357}, and T_{517} are all contained inside T_{137}, we could then search for the point inside these smaller triangles. Therefore, we can add some additional lines of code to Algorithm 11 for updating a given history graph $\mathcal{G}$. In particular, we define the following new method for Algorithm 11:

function SplitTriangle(i, j, k, r, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$, $\mathcal{G}$)
    `SplitTriangle`(i, j, k, r, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
    `AddNode`($\mathcal{G}$, T_{ijr}, T_{jkr}, T_{kir})
    `AddEdge`($\mathcal{G}$, T_{ijk}, T_{ijr}, T_{jkr}, T_{kir})
end function

Here, `AddNode`($\mathcal{G}$, T_1, T_2, $\ldots$) adds the nodes $T_1, T_2, \ldots$ into the graph $\mathcal{G}$, and `AddEdge`($\mathcal{G}$, T, V_1, V_2, $\ldots$) adds the nodes $V_1, V_2, \ldots$ into the set of out-neighbours of T.

We can apply similar ideas to `AddTriangle`, `FlipEdge` and `SplitEdge`, as given below.

function AddTriangle(i, j, k $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$, $\mathcal{G}$)
    `AddTriangle`(i, j, k, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
    `AddNode`($\mathcal{G}$, T_{ijk})
end function

function FlipEdge(i, j, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$, $\mathcal{G}$)
    `FlipEdge`(i, j, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
    $\ell = \mathcal{A}(e_{ij})$
    $k = \mathcal{A}(e_{ji})$
    `AddNode`($\mathcal{G}$, $T_{\ell kj}$, $T_{\ell ik}$)
    `AddEdge`($\mathcal{G}$, T_{ikj}, $T_{\ell kj}$, $T_{\ell ik}$)
    `AddEdge`($\mathcal{G}$, $T_{ij\ell}$, $T_{\ell kj}$, $T_{\ell ik}$)
end function

function SplitEdge(i, j, r, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$, $\mathcal{G}$)
    `SplitEdge`(i, j, r, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
    `AddNode`($\mathcal{G}$, T_{irk}, T_{rjk})
    `AddEdge`($\mathcal{G}$, T_{ijk}, T_{irk}, T_{rjk})
end function

We also define a method for `LegaliseEdge` that includes the argument $\mathcal{G}$ and makes use of the new `FlipEdge` that now includes an argument $\mathcal{G}$.

Now let us describe how we use all these ideas for point location. Since $\mathcal{G}$ is acyclic, there are no loops, and so we can search down the graph, noting that the leaf nodes of the graph are the current triangles in the triangulation, and these leaf nodes have out degree 0. So, we can recursively search down the DAG until finding a leaf node with out degree 0. We will know once we have reached this node that we have found the triangle in the current triangulation that contains the point p_r, as long as we only go down nodes that contain the point p_r already. This is summarised in Algorithm 17. We note that this procedure could be improved for example with the work of Kolingerová and Žalik (2002) who make greater use of the geometric information available from the history provided from the DAG.

Algorithm 17 Finding which triangle contains a point using the history graph.

Inputs:
- A point p_r.
- The history graph $\mathcal{G}$, the point set $\mathcal{P}$, and an initial node T_{ijk}.

Outputs:
- The triangle T in the current triangulation that contains p_r, and a flag q such that $q = 1$ if p_r is in the interior of T, or $q = 0$ if p_r is on an edge of T.

function LocateTriangle($\mathcal{G}$, $\mathcal{P}$, r, i, j, k)
  $\mathcal{O}$ = `OutNeighbours`($\mathcal{G}$, T_{ijk}) ▷ `OutNeighbours` gets the set of out neighbours
  $\mathcal{O} == \emptyset$ `&&` **return** T_{ijk}, `IsInTriangle`(i, j, k, $\mathcal{P}$, r)
  for $V_{abc} \in \mathcal{O}$ **do** ▷ There will be 3 triangles in $\mathcal{O}$ at most
    `IsInTriangle`(a, b, c, $\mathcal{P}$, r) ≥ 0 `&&` **return** `LocateTriangle`($\mathcal{G}$, $\mathcal{P}$, r, a, b, c)
  end for
end function

The algorithm

Now we can give the algorithm itself. This algorithm is given in Algorithm 18. The algorithm starts by computing the coordinates of the super triangle's vertices, and then initialises all the data structures based on this super triangle. A random insertion order is then obtained by selecting a random permutation of the indices $\{1, \ldots, n\}$. With this insertion order, we then loop over each index and add points to the triangulation one at a time, first finding the triangle that contains the point and splitting the triangle (and edge) about the point p_r. At the end of the rth loop, we have the Delaunay triangulation of $\{p_{-1}, p_{-2}, p_{-3}, p_{v_1}, p_{v_2}, \ldots p_{v_r}\}$. When we have added all n points, the function `RemoveSuperTriangle` in Algorithm 19 is used to delete all triangles that have one of the super triangle's coordinates as one of its vertices. This is done by looping over all the edges e_{uv} such that $T_{uvw} \in \mathcal{T}$, where $w \in \{-1, -2, -3\}$, making use of the adjacent-to-vertex map. In this function, we avoid the use of `DeleteTriangle` to avoid issues with boundary edges, noting that a primary task of this function is to repair the convex hull from the original super triangle's boundary. The final result is thus $\mathcal{DT}(\{p_1, \ldots, p_n\})$.

1.6.2 Bowyer-Watson algorithm

We now discuss the Bowyer-Watson algorithm, introduced by Bowyer (1981) and Watson (1981). This algorithm works similarly to de Berg's method, with points being inserted one at a time, but the way the triangulation is updated following the insertion of a point is different. In this method, when a point is added into the triangulation we delete all

Algorithm 18 Computing the Delaunay triangulation with de Berg's algorithm.

Inputs:

- A point set $\mathcal{P} = [p_1, \ldots, p_n]$.

Outputs:

- A Delaunay triangulation $\mathcal{T}$ with adjacent map $\mathcal{A}$, adjacent-to-vertex map $\mathcal{A}^{-1}$, graph $\mathcal{N}$, and history graph $\mathcal{G}$.

```
function DelaunayTriangulationBerg(𝒫)
    Compute the coordinates p_{-1}, p_{-2}, p_{-3} of the super triangle T_{-1,-2,-3} using (1.1).
    Initialise 𝒯, 𝒜, 𝒜^{-1}, 𝒩, and 𝒢 as empty data structures.
    AddTriangle(−1, −2, −3, 𝒯, 𝒜, 𝒜^{-1}, 𝒩, 𝒢)
    Generate a permutation {v_1, ..., v_n} of {1, ..., n}.    ▷ Randomised insertion order.
    for r ∈ {v_1, ..., v_n} do
        T_{ijk}, q = LocateTriangle(𝒢, 𝒫, r, T_{-1,-2,-3})
        if q == 1 then                                ▷ p_r is in the interior of T_{ijk}
            SplitTriangle(i, j, k, r, 𝒯, 𝒜, 𝒜^{-1}, 𝒩, 𝒢)
            LegaliseEdge(i, j, r, 𝒯, 𝒜^{-1}, 𝒩, 𝒫, 𝒢)
            LegaliseEdge(j, k, r, 𝒯, 𝒜^{-1}, 𝒩, 𝒫, 𝒢)
            LegaliseEdge(k, i, r, 𝒯, 𝒜^{-1}, 𝒩, 𝒫, 𝒢)
        else if q == 0 then                           ▷ p_r is on an edge of T_{ijk}
            e_{ij} = FindEdge(T_{ijk}, 𝒫, r)          ▷ p_r is on the edge e_{ij} of T_{ijk}
            k = 𝒜(e_{ij})
            ℓ = 𝒜(e_{ji})
            SplitEdge(i, j, r, 𝒯, 𝒜, 𝒜^{-1}, 𝒩, 𝒢)
            if !IsBoundaryEdge(j, i, 𝒜) then
                SplitEdge(j, i, r, 𝒯, 𝒜, 𝒜^{-1}, 𝒩, 𝒢)
            end if
            LegaliseEdge(i, ℓ, r, 𝒯, 𝒜^{-1}, 𝒩, 𝒫, 𝒢)
            LegaliseEdge(ℓ, j, r, 𝒯, 𝒜^{-1}, 𝒩, 𝒫, 𝒢)
            LegaliseEdge(j, k, r, 𝒯, 𝒜^{-1}, 𝒩, 𝒫, 𝒢)
            LegaliseEdge(k, i, r, 𝒯, 𝒜^{-1}, 𝒩, 𝒫, 𝒢)
        end if
    end for
    RemoveSuperTriangle(𝒯, 𝒜, 𝒜^{-1}, 𝒩)
    return 𝒯, 𝒜, 𝒜^{-1}, 𝒩, 𝒢
end function
```

the triangles whose circumdisks contain the point, thus evacuating a polygonal cavity in the triangulation. The triangulation is then repaired by connecting the boundaries of this polygonal cavity to the new point. Figure 1.3 gives an example of the procedure, with Figure 1.12a showing the existing triangulation and the point p_{11} to be added. We first find all triangles whose open circumdisks contain p_{11}, as these triangles are no longer Delaunay. We show these triangles in blue in Figure 1.12b. These triangles all need to be deleted, evacuating the blue polygonal cavity from Figure 1.12b; note that the guarantee that this blue region is a polygonal cavity is given by Proposition 1.1. With this region now deleted, we connect the vertices of the polygonal cavity to the new point p_{11}. These new edges are shown in blue in Figure 1.12c. The two propositions below guarantee that the evacuated cavities are star-shaped, allowing us to guarantee that all triangles

Algorithm 19 Removing the super triangle from Algorithm 18.

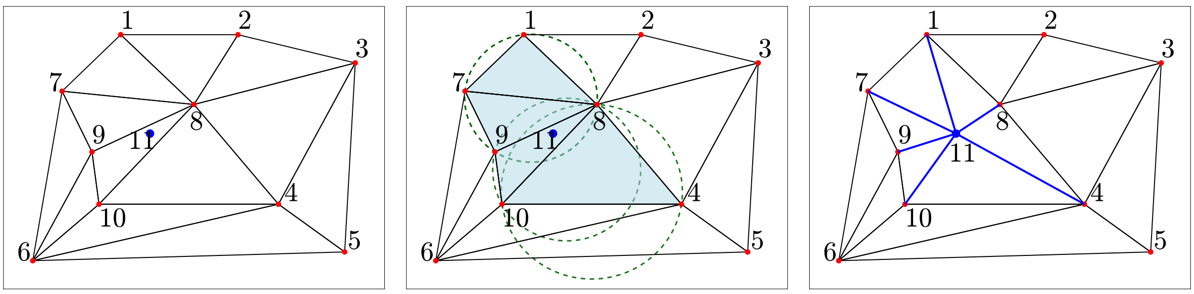
Inputs:

- A triangulation $\mathcal{T}$, adjacent map $\mathcal{A}$, adjacent-to-vertex map $\mathcal{A}^{-1}$, and graph $\mathcal{N}$.

Outputs:

- An updated triangulation $\mathcal{T}$ with the super triangle removed.

function RemoveSuperTriangle($\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$)
    for $w \in \{-1, -2, -3\}$ **do**
        for $e_{uv} \in \mathcal{A}^{-1}(w)$ **do**
            `delete!`($\mathcal{A}$, e_{wu}, e_{wv}, e_{uw}, e_{vw})
            `delete!`($\mathcal{A}^{-1}(u)$, e_{vw})
            `delete!`($\mathcal{A}^{-1}(v)$, e_{wu})
            if $u \geq 1$ `&&` $v \geq 1$ **then**▷ A boundary edge must have two positive indices.
                $\mathcal{A}(e_{uv}) = \partial$
                `push!`($\mathcal{A}^{-1}(\partial)$, e_{uv})
            end if
            `delete!`($\mathcal{T}$, T_{uvw})
        end for
        `delete!`($\mathcal{N}$, w)
        `delete!`($\mathcal{A}^{-1}$, w)
    end for
    `delete!`($\mathcal{A}^{-1}(\partial)$, $e_{-1,-3}$, $e_{-3,-2}$, $e_{-2,-1}$)
end function



(a) A point to be added. (b) Triangles that are no longer Delaunay. (c) Updated triangulation with p_{11} now included.

Figure 1.12: Process for inserting a vertex inside a triangulation with the Bowyer-Watson algorithm. In (a), the point p_{11} marked in blue is to be added. In (b), we locate all triangles whose open circumdisk contains p_{11} and mark them in blue. These triangles are no longer Delaunay. (c) This is the updated triangulation, with new edges shown in blue. This figure is based on Cheng et al. (2013, Figure 3.3).

whose open circumdisks contains the point to be added will be found by looking across neighbouring triangles, and that the new added triangles are all Delaunay; see Cheng et al. (2013, Proposition 3.1, Proposition 3.2).

Proposition 1.1 (Star-shaped cavity). Let u be a point to be added inside an existing Delaunay triangulation. The union of the triangles whose open circumdisks contain u is a connected star-shaped polygon, meaning that for every point p in the polygon, the polygon includes the line segment pu. ■

Proposition 1.2 (The new edges are Delaunay)*.* Let u be a point to be added inside an existing Delaunay triangulation. Let T be a triangle that is deleted because its open circumdisk contains u. Let w be a vertex of T. Then the edge uw is strongly Delaunay. ■

Point insertion

Let us first discuss the problem of inserting points into an existing triangulation. As we discussed above, the method for adding a point is to remove all triangles whose open circumdisks contain the new point, and then to connect the vertices of the resulting evacuated polygonal cavity to the new point. How do we do this? The idea is to use a depth-first search, as we now explain using Figure 1.12.

The first step in the procedure is to delete $T_{9,10,8}$, as this is the triangle that p_{11} is inside of. We next search for more non-Delaunay triangles by walking over the neighbouring triangles. For example, if we walk across $e_{10,8}$ into $T_{8,10,4}$, we find that the circumdisk of $T_{8,10,4}$ contains p_{11}, and so we delete $T_{8,10,4}$ also. When we continue and walk over e_{48} and $e_{10,4}$, we find that the triangles $T_{8,4,3}$ and $T_{10,6,4}$ do not contain p_{11} in their respective circumdisks. Therefore, the boundary of the evacuated polygonal cavity must contain $e_{4,8}$ and $e_{10,4}$. We would then apply the same procedure to the edges $e_{9,10}$ and $e_{8,9}$ to identify the remaining edges of the polygonal cavity's boundary. We formalise this procedure in Algorithm 20, with the procedure used for searching through the cavity given by Algorithm 21. Note that Line 4 of Algorithm 21 first checks if the edge is a boundary edge, noting that a boundary edge, provided we have reached such an edge while traversing the cavity, will necessarily form the boundary of the polygonal cavity.

Algorithm 20 Inserting a vertex inside a triangulation with the Bowyer-Watson method.

Inputs:

- A vertex r to be added, known to be inside the triangle T_{ijk}.
- An existing triangulation $\mathcal{T}$, the adjacent map $\mathcal{A}$, the adjacent-to-vertex map $\mathcal{A}^{-1}$, the graph $\mathcal{N}$, and the point set $\mathcal{P}$.

Outputs:

- An updated triangulation $\mathcal{T}$ that now has u added into it.

procedure AddPointBowyer(r, i, j, k, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$, $\mathcal{P}$)
    `DeleteTriangle`(i, j, k, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$; `protect_boundary = true`)
    `DigCavity`(r, i, j, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$, $\mathcal{P}$) ▷ Identify other deleted triangles and
    `DigCavity`(r, j, k, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$, $\mathcal{P}$) ▷ insert new triangles.
    `DigCavity`(r, k, i, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$, $\mathcal{P}$)
end procedure

The above discussion is only valid if the point to be added is inside the existing triangulation. What if we are adding a point outside of the triangulation? Following Cheng et al. (2013, Section 3.4), the solution is to imagine that each boundary edge has a third vertex at infinity, called a *ghost vertex*, and the resulting triangle is called a *ghost triangle*. The two edges that adjoin the ghost vertex are called *ghost edges*. With our existing data structures, notice that we can easily represent the ghost vertex by ∂, for example $\mathcal{A}(e_{ij}) = \partial$ could instead be interpreted as the ghost triangle $T_{ij\partial}$ rather than simply saying that e_{ij} is a boundary edge (note that Cheng et al. (2013) use g to denote a ghost vertex). One additional feature that we do need to add into our data structures is the addition of the ghost edges into the adjacent and adjacent-to-vertex maps, i.e. edges

Algorithm 21 Digging the cavities for the Bowyer-Watson method.

Inputs:

- A vertex r being added via Algorithm 20, and an edge e_{ij} to traverse for evacuating the polygonal cavity.
- An existing triangulation $\mathcal{T}$, the adjacent map $\mathcal{A}$, the adjacent-to-vertex map $\mathcal{A}^{-1}$, the graph $\mathcal{N}$, and the point set $\mathcal{P}$.

Outputs:

- An updated triangulation $\mathcal{T}$ that has now evacuated more of the polygonal cavity from Algorithm 20, or added a triangle onto the boundary of the cavity.

procedure DigCavity(r, i, j, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$, $\mathcal{P}$)
    $\ell = \mathcal{A}(e_{ji})$ ▷ $T_{ji\ell}$ is the triangle on the other side of the edge (i, j) from r
    $\ell == \emptyset$ `&&` **return** ▷ The triangle has already been deleted in this case.
    $\delta = \ell \neq \partial$ `&& IsInCircle`($\mathcal{P}$, r, i, j, ℓ)
    if $\delta == 1$ **then**
        `DeleteTriangle`(j, i, ℓ, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$; `protect_boundary = true`)
        `DigCavity`(r, i, ℓ, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$, $\mathcal{P}$) ▷ Recursively identify more deleted
        `DigCavity`(r, ℓ, j, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$, $\mathcal{P}$) ▷ triangles and insert new triangles.
    else ▷ e_{ij} is an edge of the polygonal cavity in this case.
        `AddTriangle`(r, i, j, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$, $\mathcal{P}$)
    end if
end procedure

of the form e_{ij} where $i = \partial$ or $j = \partial$. With this update, we note that $\mathcal{A}$ and $\mathcal{A}^{-1}$ will now truly be inverses of each other. We also now include ∂ in $\mathcal{N}$.

Ghost triangles

The issues arising from this notion of a ghost triangle require a detailed discussion. In particular, we need to address the following issues:

1. Given a ghost triangle $T_{ij\partial}$, how can we define its circumdisk?
2. Given a ghost triangle $T_{ij\partial}$, what does it mean for a point p to be in the circumdisk of $T_{ij\partial}$?
3. How can we modify our existing algorithms so that the ghost edges and ghost triangles are explicitly added into $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, and $\mathcal{N}$?
4. Given a point that is outside of the triangulation, how do we decide, out of all ghost triangles, what ghost triangle has p in inside it?
5. What changes need to be made to Algorithm 21 so that it now works for points outside the triangulation?
6. Once a triangulation has been computed, how can we efficiently remove all ghost edges and ghost triangles? This will mostly be needed for visualisation purposes – once this information has been removed, we would have to re-add it if we need to do any more with the triangulation.

Circumdisk of a ghost triangle Let us first consider the issue of defining a ghost triangle's circumdisk. Take some ghost triangle $T_{ij\partial}$, and let $C_{ij\partial}$ be a circle through p_i, p_j, and p_∂, with p_∂ denoting the ghost vertex. Since the ghost vertex is at infinity, this circle has infinite radius, meaning $C_{ij\partial}$ is a line. This line is the line ℓ_{ij} through p_i and p_j; note that this is a line not a line segment. With this definition, Algorithm 4 for `IsInCircle` is modified as follows, making use of Algorithm 7 to see if a point is to the left of a given line:

```
function IsInCircle(P, i, j, k, ℓ)
    if i == ∂ then
        return IsInOuterHalfPlane(P, j, k, ℓ)
    else if j == ∂ then
        return IsInOuterHalfPlane(P, k, i, ℓ)
    else if k == ∂ then
        return IsInOuterHalfPlane(P, i, j, ℓ)
    end if
    a_x, a_y = P(i)        ▷ P(i) returns the ith point p_i in the point set P,
    b_x, b_y = P(j)        ▷ and a_x, a_y = P(i) returns the x- and y-coordinates of p_i.
    c_x, c_y = P(k)
    d_x, d_y = P(ℓ)
    Δ = | a_x − d_x   a_y − d_y   (a_x − d_x)^2 + (a_y − d_y)^2 |
        | b_x − d_x   b_y − d_y   (b_x − d_x)^2 + (b_y − d_y)^2 |
        | c_x − d_x   c_y − d_y   (c_x − d_x)^2 + (c_y − d_y)^2 |
    return sgn(Δ)
end function
function IsInOuterHalfPlane(P, v, w, ℓ)                        ▷ u == ∂.
    e = IsLeftOfLine(P, v, w, ℓ)
    if e == 0 then
        b = PointOnSegment(P, ℓ, v, w)
        if b == 1 then
            return b
        else
            return −1
        end if
    else
        return e
    end if
end function
```

The function `PointOnSegment`$(\mathcal{P}, u, v, w)$, assuming that the points p_u, p_v, and p_w are collinear, returns 1 if p_u on the open edge e_{vw}, 0 if $p_u = p_v$ or $p_u = p_w$, and -1 otherwise. This is implemented using `sameside` from `ExactPredicates.jl` (Lairez, 2019).

Now we can address the issue of defining what it means for a point to be inside a ghost triangle's circumdisk. Since the ghost vertex will be to the left of e_{ij}, or to the left of ℓ_{ij}, we can interpret the inside of the circumdisk as being the set of points to the left of ℓ_{ij}. Thus, a point p will be inside the circumdisk of $T_{ij\partial}$ if it is to the left of ℓ_{ij}.

Updating ghost triangles in the triangulation The next issue to consider is the modification of our existing algorithms so that we explicitly represent the ghost edges and ghost triangles in $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, and $\mathcal{N}$. The two algorithms to consider in detail are

AddTriangle and DeleteTriangle. In the case of AddTriangle, the functions that need to be considered for the ghost triangles are the AddBoundaryEdges function. Whenever we update the adjacent and adjacent-to-vertex maps with the new boundaries, we just need to extend them so that we also add in the ghost triangles appropriately. For example, AddBoundaryEdgesSingle in Algorithm 9 becomes:

1: **function** AddBoundaryEdgesSingle($i, j, k, b_{ij}, b_{jk}, b_{ki}, \mathcal{T}, \mathcal{A}, \mathcal{A}^{-1}, \mathcal{N}$; `update_ghost_edges = false`)
2: $u, v, w =$ `RotateTriangle`$(b_{ij}, b_{jk}, b_{ki}, i, j, k)$
3: $\mathcal{A}(e_{uw}) = \partial$
4: $\mathcal{A}(e_{wv}) = \partial$
5: `push!`$(\mathcal{A}^{-1}(\partial), e_{uw}, e_{wv})$
6: `delete!`$(\mathcal{A}^{-1}(\partial), e_{uv})$
7: `push!`$(\mathcal{N}(\partial), w)$ ▷ u and v are already in $\mathcal{N}(\partial)$.
8: **if** `update_ghost_edges` **then** ▷ Add $T_{uw\partial}$ and $T_{wv\partial}$ and delete $T_{uv\partial}$.
9: $\mathcal{A}(e_{w\partial}) = u$
10: $\mathcal{A}(e_{\partial u}) = w$
11: $\mathcal{A}(e_{v\partial}) = w$
12: $\mathcal{A}(e_{\partial w}) = v$
13: `push!`$(\mathcal{A}^{-1}(u), e_{w\partial})$
14: `push!`$(\mathcal{A}^{-1}(w), e_{\partial u}, e_{v\partial}))$
15: `push!`$(\mathcal{A}^{-1}(v), e_{\partial w})$
16: `delete!`$(\mathcal{A}^{-1}(u), e_{v\partial})$
17: `delete!`$(\mathcal{A}^{-1}(v), e_{\partial u})$
18: `push!`$(\mathcal{T}, T_{uw\partial}, T_{wv\partial})$
19: `delete!`$(\mathcal{T}, T_{uv\partial})$
20: **end if**
21: **end function**

The function now includes $\mathcal{T}$ and $\mathcal{N}$ in its arguments. Moreover, we include the keyword update_ghost_edges in case we do not have to consider ghost nodes at all, which would be useful if we are applying de Berg's method of Algorithm 18 which still uses AddTriangle. This keyword update_ghost_edges is also put into the new AddTriangle. Note also in this code that we simplify in some cases, for example in Line 7 we only add w to $\mathcal{N}(\partial)$ rather than u, v, and w, since u and v are already boundary edges prior to the addition of T_{ijk}. We also do not need to delete any edges from $\mathcal{A}$ in this function since the previous ghost edges, $e_{v\partial}$ and $e_{\partial u}$, still exist from the new ghost triangles $T_{uw\partial}$ and $T_{wv\partial}$. Applying similar ideas for the cases of two and three boundary edges, we obtain the following new forms for AddBoundaryEdgesDouble and BoundaryEdgesTriple:

1: **function** AddBoundaryEdgesDouble($i, j, k, b_{ij}, b_{jk}, b_{ki}, \mathcal{T}, \mathcal{A}, \mathcal{A}^{-1}, \mathcal{N}$; `update_ghost_edges = false`)
2: $u, v, w =$ `RotateTriangle`$(!b_{ij}, !b_{jk}, !b_{ki}, i, j, k)$
3: $\mathcal{A}(e_{vu}) = \partial$
4: `push!`$(\mathcal{A}^{-1}(\partial), e_{vu})$
5: `delete!`$(\mathcal{A}^{-1}(\partial), e_{vw}, e_{wu})$
6: `delete!`$(\mathcal{N}(\partial), w)$ ▷ $e_{w\partial}$ was removed; u and v are still in $\mathcal{N}(\partial)$, though.
7: **if** `update_ghost_edges` **then** ▷ Add $T_{vu\partial}$ and delete $T_{vw\partial}$ and $T_{wu\partial}$.
8: $\mathcal{A}(e_{u\partial}) = v$
9: $\mathcal{A}(e_{\partial v}) = u$

delete!($\mathcal{A}$, $e_{w\partial}$, $e_{\partial w}$) ▷ This ghost edge $e_{w\partial}$ is now obstructed by e_{vu}.
push!($\mathcal{A}^{-1}(v)$, $e_{u\partial}$)
push!($\mathcal{A}^{-1}(u)$, $e_{\partial v}$)
delete!($\mathcal{A}^{-1}(u)$, $e_{\partial w}$)
delete!($\mathcal{A}^{-1}(v)$, $e_{w\partial}$)
delete!($\mathcal{A}^{-1}(w)$, $e_{u\partial}$, $e_{\partial v}$)
push!($\mathcal{T}$, $T_{vu\partial}$)
delete!($\mathcal{T}$, $T_{vw\partial}$, $T_{wu\partial}$)
end if
end function

function AddBoundaryEdgesTriple(i, j, k, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$; update_ghost_edges = false)
$\mathcal{A}(e_{ji}) = \partial$
$\mathcal{A}(e_{ik}) = \partial$
$\mathcal{A}(e_{kj}) = \partial$
push!($\mathcal{A}^{-1}(\partial)$, e_{ji}, e_{ik}, e_{kj})
push!($\mathcal{N}(\partial)$, i, j, k)
if update_ghost_edges **then** ▷ Add $T_{ik\partial}$, $T_{kj\partial}$, and $T_{ji\partial}$.
$\mathcal{A}(e_{i\partial}) = j$
$\mathcal{A}(e_{\partial j}) = i$
$\mathcal{A}(e_{j\partial}) = k$
$\mathcal{A}(e_{\partial k}) = j$
$\mathcal{A}(e_{k\partial}) = i$
$\mathcal{A}(e_{\partial i}) = k$
push!($\mathcal{A}^{-1}(i)$, $e_{\partial j}$, $e_{k\partial}$)
push!($\mathcal{A}^{-1}(j)$, $e_{i\partial}$, $e_{\partial k}$)
push!($\mathcal{A}^{-1}(k)$, $e_{j\partial}$, $e_{\partial i}$)
push!($\mathcal{T}$, $T_{ji\partial}$, $T_{ik\partial}$, $T_{kj\partial}$)
end if
end function

The ideas used for extending the AddBoundaryEdges functions can be used to extend the DeleteBoundaryEdges functions. Adding a keyword update_ghost_edges to DeleteTriangle, we now define the new methods for DeleteBoundaryEdges.

function DeleteBoundaryEdgesSingle(i, j, k, b_{ji}, b_{ik}, b_{kj}, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$; update_ghost_edges = false)
u, v, w = RotateTriangle(b_{ji}, b_{kj}, b_{ik}, i, j, k)
delete!($\mathcal{A}$, e_{vu})
delete!($\mathcal{A}^{-1}(\partial)$, e_{vu})
$\mathcal{A}(e_{vw}) = \partial$
$\mathcal{A}(e_{wu}) = \partial$
push!($\mathcal{A}^{-1}(\partial)$, e_{vw}, e_{wu})
push!($\mathcal{N}(\partial)$, w)
if update_ghost_edges **then** ▷ Add $T_{vw\partial}$ and $T_{wu\partial}$ and delete $T_{vu\partial}$.
$\mathcal{A}(e_{w\partial}) = v$
$\mathcal{A}(e_{\partial v}) = w$
$\mathcal{A}(e_{u\partial}) = w$
$\mathcal{A}(e_{\partial w}) = u$

delete!($\mathcal{A}^{-1}(v)$, $e_{u\partial}$)
delete!($\mathcal{A}^{-1}(u)$, $e_{\partial v}$)
push!($\mathcal{A}^{-1}(v)$, e_{∂})
push!($\mathcal{A}^{-1}(w)$, $e_{\partial v}$, $e_{u\partial}$)
push!($\mathcal{A}^{-1}(u)$, $e_{\partial w}$)
push!($\mathcal{T}$, $T_{vw\partial}$, $T_{wu\partial}$)
delete!($\mathcal{T}$, $T_{vu\partial}$)
end if
end function

function DELETEBOUNDARYEDGESDOUBLE(i, j, k, b_{ji}, b_{ik}, b_{kj}, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$; update_ghost_edges = false)
u, v, w = RotateTriangle(b_{ji}, b_{kj}, b_{ik}, i, j, k)
delete!($\mathcal{A}$, e_{uw}, e_{wv})
delete!($\mathcal{A}^{-1}(\partial)$, e_{uw}, e_{wv})
$\mathcal{A}(e_{uv}) = \partial$
push!($\mathcal{A}^{-1}(\partial)$, e_{uv})
delete!($\mathcal{N}(\partial)$, w)
if update_ghost_edges **then** ▷ Add $T_{uv\partial}$ and delete $T_{uw\partial}$ and $T_{wv\partial}$.
$\mathcal{A}(e_{v\partial}) = u$
$\mathcal{A}(e_{\partial u}) = v$
delete!($\mathcal{A}$, $e_{w\partial}$, $e_{\partial w}$)
delete!($\mathcal{A}^{-1}(u)$, $e_{w\partial}$)
delete!($\mathcal{A}^{-1}(v)$, $e_{\partial w}$)
delete!($\mathcal{A}^{-1}(w)$, $e_{\partial u}$, $e_{v\partial}$)
push!($\mathcal{A}^{-1}(u)$, $e_{v\partial}$)
push!($\mathcal{A}^{-1}(v)$, $e_{\partial u}$)
push!($\mathcal{T}$, $T_{uv\partial}$)
delete!($\mathcal{T}$, $T_{uw\partial}$, $T_{wv\partial}$)
end if
end function

function DELETEBOUNDARYEDGESTRIPLE(i, j, k, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$; update_ghost_edges = false)
delete!($\mathcal{A}$, e_{kj}, e_{ji}, e_{ik})
delete!($\mathcal{A}^{-1}(\partial)$, e_{kj}, e_{ji}, e_{ik})
delete!($\mathcal{N}(\partial)$, i, j, k)
if update_ghost_edges **then** ▷ Delete $T_{ji\partial}$, $T_{kj\partial}$, and $T_{ik\partial}$.
delete!($\mathcal{A}$, $e_{i\partial}$, $e_{\partial j}$, $e_{j\partial}$, $e_{\partial k}$, $e_{k\partial}$, $e_{\partial i}$)
delete!($\mathcal{A}^{-1}(j)$, $e_{i\partial}$, $e_{\partial k}$)
delete!($\mathcal{A}^{-1}(i)$, $e_{\partial j}$, $e_{k\partial}$)
delete!($\mathcal{A}^{-1}(k)$, $e_{j\partial}$, $e_{\partial i}$)
delete!($\mathcal{T}$, $T_{ji\partial}$, $T_{kj\partial}$, $T_{ik\partial}$)
end if
end function

Point location The next problem to address is that of point location. That is, if we have a point outside of the triangulation, then what ghost triangle should we say the point lives in? To answer this question, we imagine that there is some central point p_c that all ghost edges go through. For example, p_c should be the centroid of $\mathcal{P}$. We illustrate this in Figure 1.13, where we see that this choice partitions the exterior of the triangulation into separate regions for each ghost triangle. This way, we can now uniquely define the space that each ghost triangle occupies. For example, in Figure 1.13 we see that the point q is in the triangle $T_{9,10,\partial}$ as q is to the left of $e_{9,10}$, $e_{c,10}$, and $e_{9,c}$, where c refers to p_c.

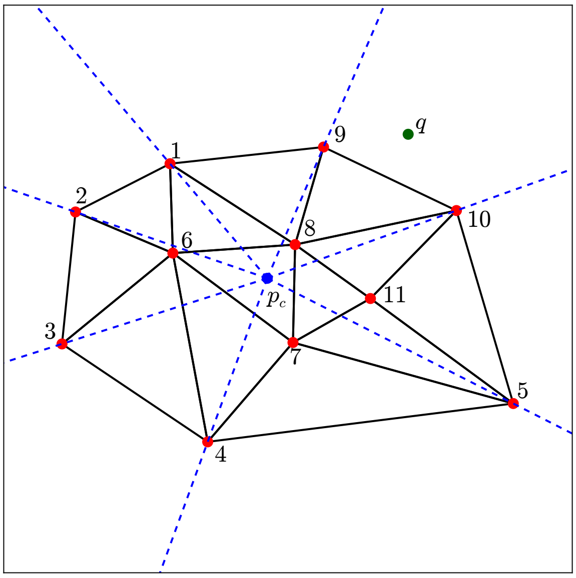


Figure 1.13: Representation of ghost triangles. The blue point p_c is at the centroid of the points, and the blue dashed lines show how the ghost edges are interpreted; the actual ghost edges are those that extend outwards from the triangulation, but we connect to the centroid to illustrate their interpretation. The point q is an example point that we see lies in $T_{9,10,\partial}$.

To start, let us address the computation of this point p_c. Suppose that $\mathcal{P} = \{p_1, \ldots, p_n\}$, so that

$$p_c = \frac{1}{n}\sum_{i=1}^{n}(x_i, y_i). \tag{1.2}$$

The issue with this definition is that our triangulations are built incrementally, meaning that the p_c should only use as many points as is currently in the triangulation anyway. So, p_c should need to be updated as we build the triangulation. One other feature of interest is that we will typically need to be access p_c whenever an algorithm attempts to access $\mathcal{P}(\partial)$. So, should we just add an extra element to $\mathcal{P}$ and let it be indexed via ∂? This is one solution, but it may cause issues with how we enumerate our points in code and we would have to change a lot of algorithms to work with it. Therefore, we will instead define a mutable point $(p_{c,x}, p_{c,y})$ that we update whenever we add or remove a point, and this point can be represented as a constant so that we do not need to add it to $\mathcal{P}$ nor do we have to modify any of our existing functions, allowing $\mathcal{P}(\partial)$ to be mapped to p_c without actually storing p_c in $\mathcal{P}$. The mutable point $(p_{c,x}, p_{c,y})$ in JULIA is represented as a mutable `Tuple` using `MutableNamedTuples.jl` (Protter, 2021). Notice, though, that (1.2) would slow down the algorithm significantly if we had to constantly re-sum all the terms, so we need a method for computing p_c^{n+1} given p_c^n and a new point $p_{n+1} = (x_{n+1}, y_{n+1})$, where $p_c^m = m^{-1}\sum_{i=1}^m (x_i, y_i)$. In fact, computing p_c^m takes $m+1$ operations to compute, and if we have n points in our final triangulation we will have computed $p_c^1, p_c^2, \ldots, p_c^n$, thus taking $\sum_{m=1}^n (m+1) = n(n+3)/2 = \mathcal{O}(n^2)$ time to compute.

Since our algorithm for computing the triangulation takes $\mathcal{O}(n \log n)$ to compute (Cheng et al., 2013, Theorem 3.7), we will have completely dominated the triangulation time by computing this simple expression (1.2). To avoid this cost, notice that

$$(n+1)p_c^{n+1} = \sum_{i=1}^{n+1}(x_i, y_i) = \sum_{i=1}^{n}(x_i, y_i) + (x_{n+1}, y_{n+1}) = np_c^n + (x_{n+1}, y_{n+1}),$$

so $p_c^{n+1} = (n+1)^{-1}(np_c^n + p_{n+1})$. Notice that the inverse of this formula, $p_c^n = n^{-1}[(n+1)p_c^{n+1} - p_{n+1}]$, could be used for updating p_c after deleting a point (as we discuss later). Thus, computing p_c over the cost of a single triangulation takes only $\mathcal{O}(n)$ time.

Now let us ensure we can correctly identify if a point is in a given ghost triangle; later we will discuss modifications to Algorithm 16. Since our method for seeing if a point is in a triangle just makes use of `IsLeftOfLine`, we only have to modify Algorithm 7. See that if we want to see if a point is to the left of $e_{i\partial}$, then this is the same as seeing if a point is to the left of e_{ci}. Similarly, to see if a point is to the left of $e_{\partial j}$, we just see if it is to the left of e_{jc}. Since p_c is accessed through $\mathcal{P}(\partial)$, we see that the coordinates of the edge have simply been swapped in these ghost edge cases, with the ghost vertex at infinity moved to the vertex. We thus obtain the following modification to `IsLeftOfLine`, which automatically makes `IsInTriangle` work also:

1: **function** IsLeftOfLine($\mathcal{P}$, i, j, k)
2: $(i == -1$ `&&` $j == -3)$ `&&` **return** -1
3: $(i == -1$ `&&` $j == -2)$ `&&` **return** 1
4: $(i == -3$ `&&` $j == -1)$ `&&` **return** 1
5: $(i == -3$ `&&` $j == -2)$ `&&` **return** -1
6: $(i == -2$ `&&` $j == -3)$ `&&` **return** 1
7: $(i == -2$ `&&` $j == -1)$ `&&` **return** -1
8: **if** $i == \partial$ `||` $j == \partial$ **then**
9: $j, i = i, j$ ▷ Swap i and j.
10: **end if**
11: **return** `IsOriented`($\mathcal{P}(i)$, $\mathcal{P}(j)$, $\mathcal{P}(k)$) ▷ $\mathcal{P}(\partial)$ will get mapped to p_c.
12: **end function**

Notice that we only had to add an extra case to this code, amounting to only three lines in Lines 8–10, to make these predicates `IsLeftOfLine` and `IsInTriangle` work.

Now let us discuss how we can modify Algorithm 16 so that we can correctly work with ghost triangles. The main issue in this case is that the point location algorithm makes the assumption that we will eventually reach an edge that goes beyond q, as this is when the orientation of the points will swap so that the condition in Line 25 in Algorithm 16 becomes false. Moreover, the selection of the initial triangle needs to be modified for the case where a point is outside the triangulation.

To discuss our solution to this problem, let us first give the modified form of Algorithm 16 and then we will discuss all the new components.

This new algorithm is given in Algorithm 22. The first modification is Line 2 which checks if the initial vertex p_k is a boundary point, or if the triangulation contains ghost triangles. The function `IsBoundaryPoint` checks if p_k is a boundary point, defined by:

function IsBoundaryPoint(u, $\mathcal{A}$, $\mathcal{N}$)
if $\partial \in \mathcal{N}$ **then** ▷ More efficient method if the triangulation has ghost triangles.
return $u \in \mathcal{N}(\partial)$
else ▷ If the triangulation has no ghost triangles.

Algorithm 22 Point location with the jump-and-march algorithm, updated to work with ghost triangles.

Inputs:

- A vertex k to start the walk at and a query point q. See also Algorithm 15 for choosing this vertex k randomly.
- An adjacent map $\mathcal{A}$, adjacent-to-vertex map $\mathcal{A}^{-1}$, and point set $\mathcal{P}$.

Outputs:

- A triangle T_{ijk} that contains q in its interior.

function JumpAndMarch(k, q, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{P}$)
    if `!IsBoundaryPoint`(k, $\mathcal{A}$, $\mathcal{N}$) `||` `!HasGhostTriangles`($\mathcal{A}$, $\mathcal{A}^{-1}$) **then**
        p, e_{ij}, p_i, p_j = `SelectInitialTriangle`(q, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{N}$, k, $\mathcal{P}$)
    else
        e_{ij}, e_{edge}, e_{tri} = `CheckInteriorEdgeIntersections`(q, $\mathcal{A}$, $\mathcal{N}$, k, $\mathcal{P}$)
        if e_{tri} **then**
            return T_{ijk}
        else if `!`e_{edge} **then**
            e_{ij} = `StraightLineSearchGhostTriangles`(q, $\mathcal{A}$, k, $\mathcal{P}$)
            return $T_{ij\partial}$
        end if
        $p, p_i, p_j = \mathcal{P}(k), \mathcal{P}(i), \mathcal{P}(j)$
    end if
    while `IsOriented`(p_i, p_j, q) $== 1$ **do**
        $k = \mathcal{A}(e_{ij})$
        if $k == \partial$ **then**
            if `HasGhostTriangles`($\mathcal{A}$, $\mathcal{A}^{-1}$) **then**
                $e_{i'j'}$ = `StraightLineSearchGhostTriangles`(q, $\mathcal{A}$, i, $\mathcal{P}$)
                return $T_{i'j'\partial}$
            else
                return `JumpAndMarch`(i, q, $\mathcal{A}$, $\mathcal{A}^{-1}$, $\mathcal{P}$)
            end if
        end if
        $p_k = \mathcal{P}(k)$
        if `IsOriented`(p, q, p_k) $== -1$ **then**
            $j = k$
            $p_j = p_k$
        else
            $i = k$
            $p_i = p_k$
        end if
    end while
    $k = \mathcal{A}(e_{ji})$
    return T_{jik}
end function

---

        for $v \in \mathcal{N}(u)$ **do** ▷ Try and find a boundary edge with u as an endpoint.
            `IsBoundaryEdge`(u, v, $\mathcal{A}$) `&&` **return** `true`
        end for

        return `false`
    end if
end function

The function `HasGhostTriangles` checks if the triangulation contains ghost triangles, and is defined by:

function HasGhostTriangles($\mathcal{A}$, $\mathcal{A}^{-1}$)
    e_{uv} = `iterate`($\mathcal{A}^{-1}(\partial)$) ▷ Pick some edge from $\mathcal{A}^{-1}(\partial)$.
    return `EdgeExists`($e_{v\partial}$, $\mathcal{A}$) ▷ If e_{uv} and $e_{v\partial}$ exist, then $T_{uv\partial}$ exists.
end function

We need these checks because the behaviour will depend on whether or not a triangulation has ghost triangles; triangulations computed using the Bowyer-Watson algorithm will, but those with de Berg's method will not. If the triangulation does have ghost edges, then the behaviour for a boundary point differs from that for an interior point, with the latter case being identical to the case where a triangulation has no ghost triangles. The function `SelectInitialTriangle` in Line 3 is simply Lines 2–24 from the original algorithm in Algorithm 16.

Lines 5–13 now consider the case where the triangulation has ghost triangles and the initial vertex is on the boundary. The first step when starting from the boundary is to see if the point is in the interior of the triangulation or in the exterior, since if it is in the interior then we can just find some initial edge and use Algorithm 16 as usual. We do this check in Line 5 using the `CheckInteriorEdgeIntersections` function, which returns e_{ij}, e_{edge}, and e_{tri}. This edge e_{ij} will be the interior edge that the line $\overrightarrow{p_kq}$ intersects, or $e_{\emptyset\emptyset}$ (an empty edge) if no such edge exists; e_{edge} is a Boolean that will records whether or not $\overline{p_kq}$ intersects an interior edge, i.e. `!`e_{edge} would mean that q is outside of the triangulation; e_{tri} is a Boolean that records whether or not the point q is in one of the solid triangles (a non-ghost triangle) neighbouring p_k, so that T_{ijk} contains q, as we return in Line 7. If `!`e_{edge} is true, then we compute a boundary edge e_{ij} such that the ghost triangle $T_{ij\partial}$ contains q, done using the function `StraightLineSearchGhostTriangles` which does the equivalent of Algorithm 16 except for ghost triangles. If e_{tri} and `!`e_{edge} are both false, then we just compute the initial points in Line 12 and proceed as usual, since q is in the interior. This function `CheckInteriorEdgeIntersections` in Line 5 is defined as follows:

function CheckInteriorEdgeIntersections(q, $\mathcal{A}$, $\mathcal{N}$, k, $\mathcal{P}$)
    $p = \mathcal{P}(k)$
    $i = \mathcal{A}(e_{k\partial})$ ▷ This is to the left of p.
    $p_i = \mathcal{P}(i)$
    o_1 = `IsOriented`(p, q, p_i) ▷ $o_1 = 1$ if p_i is left of $\overrightarrow{pq}$.
    for $r \in \{1, \ldots, |\mathcal{N}(k)| - 2\}$ **do** ▷ $|\mathcal{N}(k)|$ neighbours, two are ∂ and i.
        $j = \mathcal{A}(e_{ki})$
        $p_j = \mathcal{P}(j)$
        o_2 = `IsOriented`(p, q, p_j) ▷ $o_2 = 1$ if p_j is left of $\overrightarrow{pq}$.
        if $o_1o_2 == -1$ **then** ▷ Possible intersection.
            if `SegmentsMeet`(p, q, p_i, p_j) $== 1$ **then** ▷ Do $\overrightarrow{pq}$ and $\overrightarrow{p_ip_j}$ intersect?
                return e_{ji}, `true`, `false` ▷ Switch i and j so that p_i is left of $\overrightarrow{pq}$.
            else if `IsOriented`(p_j, p, q) $== 1$ `&&` `IsOriented`(p, p_i, q) $== 1$ **then**
                return e_{ij}, `false`, `true` ▷ No intersection, but is inside T_{ijk}.
            else
                return $e_{\emptyset\emptyset}$, `false`, `false`

17: **end if**
18: **end if**
19: $o_1,\ i,\ p_i = o_2,\ j,\ p$
20: **end for**
21: **return** $e_{\emptyset\emptyset}$, `false`, `false`
22: **end function**

This function starts by taking the point p_i that is to the left of the point p_k, as done in Lines 2–4. We then see in Line 5 if p_i is to the left of $\overrightarrow{pq}$. Then, looping over the $|\mathcal{N}(k) - 2|$ remaining neighbours of p_k, we rotate right around p_k until we find an edge e_{ij} such that p_i is left of $\overrightarrow{pq}$ and p_j is right of $\overrightarrow{pq}$, or vice versa. This case means that $o_1o_2 = -1$, as we test in Line 10. If $o_1o_2 = -1$ is true, then there are two cases:

1. First, the edge e_{ij} may actually intersect $\overrightarrow{pq}$, in which case we can start the straight line search from this edge e_{ij}. This test is done using `SegmentsMeet`, which is the function `meet` from `ExactPredicates.jl` (Lairez, 2019), and returns 1 if the two open line segments intersect in a single point, 0 if the two closed line segments intersect in one or several points, and -1 otherwise.

2. The second case is that the edge e_{ij} does not intersect $\overrightarrow{pq}$, which could mean that q is on the other side of p away from e_{ij}, or q is on the same side as p but still away from e_{ij}, meaning it must be inside the triangle T_{ijk}. To differentiate between these two cases, we first see if q is left of e_{jk} and left of e_{ki}, as we test in Line 13, as this implies that q must be inside T_{ijk}. If this test is not true, then this failure along with the first case's failure imply that q is away from p_k relative to the edge e_{ij}, which must mean that q is outside of the triangulation as p_k is a boundary point, and so we return in Line 16.

If we never find an intersection or a triangle containing q, then the point q must be outside of the triangulation, and so we return in Line 21.

Next, the function for jumping over ghost triangles `StraightLineSearchGhostTriangles` in Line 9 of Algorithm 22 is defined by:

function StraightLineSearchGhostTriangles(q, $\mathcal{A}$, k, $\mathcal{P}$)
  $p_c = \mathcal{P}(\partial)$ ▷ This is the centroid of $\mathcal{P}$, recall.
  $i = k$
  $p_i = \mathcal{P}(k)$
  $o_{ciq} =$ `IsOriented`(p_c, p_i, q)
  if $o_{ciq} == 1$ **then** ▷ q is left of the ghost edge through p_k, so rotate left.
    $j = \mathcal{A}(e_{i\partial})$
    $p_j = \mathcal{P}(j)$
    while `IsOriented`(p_c, p_j, q) $== 1$ **do** ▷ Until we find an intersection.
      $i = j$
      $p_i = p_j$
      $j = \mathcal{A}(e_{i\partial})$
      $p_j = \mathcal{P}(j)$
    end while
    return e_{ji} ▷ Swap the orientation so that e_{ij} is a boundary edge.
  else ▷ q is right of the ghost edge through p_k, so rotate right.
    $j = \mathcal{A}(e_{\partial i})$
    $p_j = \mathcal{P}(j)$

19: **while** `IsOriented`$(p_c, p_j, q) == -1$ **do**
20: $i = j$
21: $p_i = p_j$
22: $j = \mathcal{A}(e_{\partial i})$
23: $p_j = \mathcal{P}(j)$
24: **end while**
25: **return** e_{ij}
26: **end if**
27: **end function**

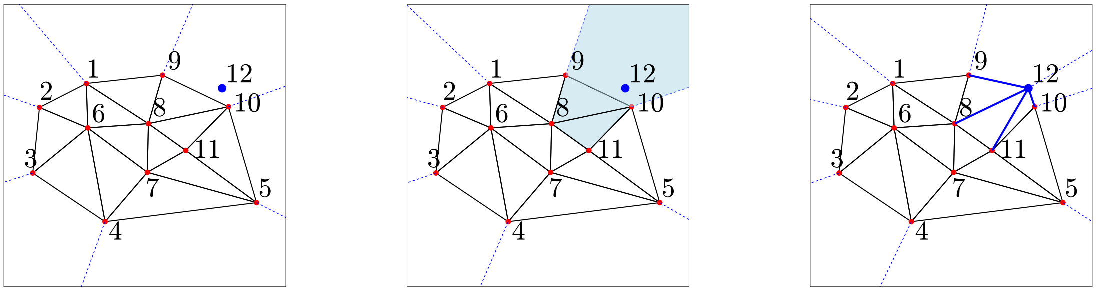
This function follows a very similar idea to that of a straight line search that we use in Algorithm 16. We first determine if q is left or right of the ghost edge through p_k, remembering that the ghost edge is interpreted as passing through the centroid p_c and the boundary point. If q is left of this ghost edge, then we should rotate left around the boundary until we get a ghost edge that goes past q, as this will tell us that we have passed a ghost triangle containing q. This left rotation is done in Lines 6–15, and Line 9 is what tells us to stop rotating. Lines 16– 26 performs the right rotation. In this function, we only return a boundary edge e_{ij} since the third vertex of the ghost triangle will simply be ∂.

The remainder of Algorithm 22 is mostly unchanged, except we need to check for the case where our straight line search takes us into a boundary edge, in which case the point q is outside of the triangulation. We check this in Line 16, where we perform a straight line search over the ghost triangles once we go past such an edge.

Adding a point outside of the triangulation

Now that we have an understanding of ghost triangles, we can consider what happens when we try to add a point outside of a triangulation. We recall that a point p is inside the circumdisk of a ghost triangle $T_{ij\partial}$ if p is to the left of the line ℓ_{ij}. So, there are two cases where a ghost triangle $T_{ij\partial}$ needs to be deleted after the insertion of a point p, meaning e_{ij} is no longer a boundary edge. Following Cheng et al. (2013, Section 3.4), the first case is where p is to the left of ℓ_{ij} away from the triangulation, and the second case is where p is on e_{ij}, in which case we should split edge into two boundary edges and also contain with $\mathcal{A}(e_{ji})$ (similar to Algorithm 18's method for handling collinear points). We call the union of the set of points left of ℓ_{ij} and those on e_{ij} the *outer halfplane* of e_{ij}; note that the outer halfplane is neither open nor closed.

Figure 1.14 illustrates what happens when we add a point outside of the boundary. It turns out that with the modifications to our existing algorithms, Algorithm 20 works with no changes.



(a) A point to be added.

(b) Triangles that are no longer Delaunay. The unbounded triangle is the ghost triangle.

(c) Updated triangulation with p_{12} now included.

Figure 1.14: Process for inserting a vertex out a triangulation with the Bowyer-Watson algorithm. In (a), the point p_{12} marked in blue is to be added. In (b), we locate all triangles whose open circumdisk contains p_{12} and mark them in blue. These triangles are no longer Delaunay. The ghost triangle $T_{9,10,\partial}$, represented by the unbounded triangle, has p_{12} in its open circumdisk as it is left of $e_{9,10}$. (c) This is the updated triangulation, with new edges shown in blue. In each figure, the dashed lines are to be interpreted as the ghost edges.

Bibliography

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, *59*(1), 65–98.

Bowyer, A. (1981). Computing Dirichlet tessellations. *Computer Journal*, *24*, 162–166.

Cheng, S.-W., Dey, T. K., & Shewchuk, J. R. (2013). *Delaunay mesh generation*. CRC Press.

Chew, L. P. (1990). *Building Voronoi diagrams for convex polygons in linear expected time* (Technical Report PCS-TR90-147). Department of Mathematics and Computer Science, Dartmouth College. Hanover, New Hampshire.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms* (4th ed.). MIT Press.

de Berg, M., Cheong, O., van Kreveld, M., & Overmars, M. (2008). *Computational geometry: Algorithms and applications* (3rd ed.). Springer.

de Berg, M., van Kreveld, M., Overmars, M., & Schwarzkopf, O. (1999). *Computational geometry: Algorithms and applications* (2nd ed.). Springer.

Deo, N. (2018). Graphs. In D. P. Mehta & S. Sahni (Eds.), *Handbook of data structures and algorithms* (2nd ed., pp. 49–66). CRC Press.

Devillers, O., Pion, S., & Teillaud, M. (2002). Walking in a triangulation. *International Journal of Foundations of Computer Science*, *13*, 181–199.

Devroye, L., Mücke, E., & Zhu, B. (1998). A note on point location in Delaunay triangulations of random points. *Algorithmica*, *22*, 477–482.

Kolingerová, I., & Žalik, B. (2002). Improvements to randomized incremental Delaunay insertion. *Computers & Graphics*, *26*, A477–490.

Lairez, P. (2019). ExactPredicates.jl [v2.2.2. Accessed on 2022 October 9.].

Lin, D. (2013). DataStructures.jl [v0.18.13. Accessed on 2022 October 6.].

Mehta, D. P. (2018). Trees. In D. P. Mehta & S. Sahni (Eds.), *Handbook of data structures and algorithms* (2nd ed., pp. 35–48). CRC Press.

Mücke, E. P., Saias, I., & Zhu, B. (1999). Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations. *Computational Geometry*, *12*, 63–83.

Protter, M. (2021). MutableNamedTuples.jl [v0.1.2. Accessed on 2022 October 21.].

Scheinerman, E. (2014). SimpleGraphs.jl [v0.8.3. Accessed on 2022 October 6.].

Shewchuk, J. R. (1996). Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator [From the First ACM Workshop on Applied Computational Geometry]. In M. C. Lin & D. Manocha (Eds.), *Applied computational geometry: Towards geometric engineering* (pp. 203–222, Vol. 1148). Springer-Verlag.

Watson, D. F. (1981). Computing the n-dimensional Delaunay tessellation with application to Voronoi polytyopes. *Computer Journal*, *24*, 167–172.