Chapter 1

Algorithm for R between two arbitrary points in a mesh

To measure the Dirichlet energy, we need to calculate the rotation coefficient R between two arbitrary points q and p that do not necessarily lie within the same tetrahedron. Since the metric and curl is different in each tet, we need to be able to efficiently determine all tets that get intersected by the straight line from q to p, and use the correct metric for each corresponding line segment. The calculation for the coefficient then works in the following way:

Algorithm 1 Rotation coefficient R between q and p

Input (q, p)
    //returns all tets intersected by the line $\vec{qp}$ with the line segments within them
    LINESEGMENTS $\leftarrow$ *tetFinder*(q, p)
    $R \leftarrow \mathrm{Id}$
    for each SEGMENT **in** LINESEGMENTS
        $R \leftarrow R \cdot$ *calcCoeff*(SEGMENT)
return R

The missing component here is how to efficiently find all tetrahedra that get intersected. One possibility would be to use ray-triangle intersection and test against the whole mesh, but this is not practical, as we have have local information that we can exploit.

We use the idea of the straight walk from *Walking in a Triangulation*[**?**], which relies only on so called *orientation tests* to determine which triangles we traverse. This chapter covers how the *tetFinder* algorithm works, how it is made robust against degenerate cases and a full description of the algorithm is given, more than what is described in the reference.

1.1 Framework

Let $\mathcal{T}$ be a triangulation of a domain Ω that is convex. The straight walk traverses all triangles that get intersected by the line segment from q to p. The algorithm first makes an initialization step to get into a valid state, then the straight walk can start. To get a feeling how the algorithm works, let us go through an example in 2D. If the algorithm was in a valid state before, the line from q to p intersects with some edge $\vec{lr}$. Two triangles share this edge. We test on which side point p lies of this edge to decide whether the walk continues. If the walk continues, we jump through the edge to hop from the old triangle to a new one. This triangle is defined by three vertices (l, r, s). We decide if the new candidate point s lies on the left side or right side of the line from q to p. If s lies on the left, point l is moved, else point r is moved. A new edge intersected with the ray $\vec{qp}$ is found and the walk repeats. This process is illustrated in Figure 1.1

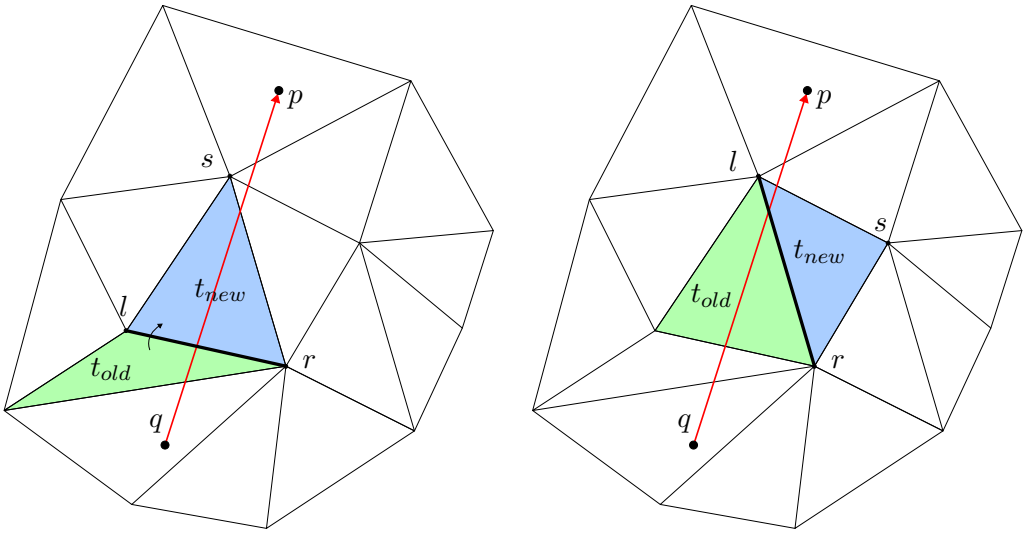

Figure 1.1. Straight walk step

Notice how the ray $\vec{qp}$ always intersects the edge $\vec{lr}$ at each update step. We can use this observation to add each edge at each update step to a list. When the algorithm terminates, we can just iterate over this list, find the intersection point of the ray $\vec{qp}$ with the edge and calculate the rotation coefficient for this segment. The straight walk in 3d works similarly. The initialization step consists of finding a starting tet t where q is contained. Then, we find the face of the tet t that gets intersected by the ray $\vec{qp}$. Again, at each step, we know that the ray goes out of our current tet t through some face e defined by vertices uvw. We decide if the walk continues by checking on which side p lies relative to e. If the walk should continue, we hop through e to a new tet t_{new}. With two orientation tests we decide which of the vertices u, v, w gets moved to the new candidate point s. This defines a new face e_{new} where our ray intersects and the walk repeats. Degenerate cases such as when the ray $\vec{qp}$ goes exactly through a vertex or when the ray lies within a face, the algorithm may get into an invalid state, where the algorithm may then traverse through cells that do not get intersected by the ray. We need to detect and escape those degenerate cases through some additional checks. These are not described in the paper[**?**]. How we handle these degenerate cases is described in Sec. 1.2.4.

1.1.1 A note on orientation tests

To determine on which side some point s lies relative to two other points q and p (that represent a line) in 2D, the geometric *orientation* predicate is used. It corresponds to evaluating the sign of a determinant. Analogously in 3D, the orientation predicate tests whether a fourth point lies above or below a plane defined by three other points. How "above" the plane is defined depends on the ordering within the determinant. In the case here, it is above if point a sees the triangle bcd when turning counterclockwise (see figure 1.2).

$$orientation(\alpha, \beta, \gamma) = sign\left(\begin{vmatrix} \beta_x - \alpha_x & \gamma_x - \alpha_x \\ \beta_y - \alpha_y & \gamma_y - \alpha_y \end{vmatrix}\right)$$

$$orientation(\alpha, \beta, \gamma, \delta) = sign\left(\begin{vmatrix} \beta_x - \alpha_x & \gamma_x - \alpha_x & \delta_x - \alpha_x \\ \beta_y - \alpha_y & \gamma_y - \alpha_y & \delta_y - \alpha_y \\ \beta_z - \alpha_z & \gamma_z - \alpha_z & \delta_z - \alpha_z \end{vmatrix}\right)$$

It is important that the sign of the determinant is evaluated exactly. If geometric predicates are implemented with floating point arithmetic, the answers may be inconsistent and wrong. Because of the finite mantissa in floating point representation, many numbers cannot be represented exactly. The machine needs to round a number x to its nearest number that is exactly representable. Many times, roundoff errors occur and the result is not exact. A famous example of this phenomenon is $0.1 + 0.2 =$

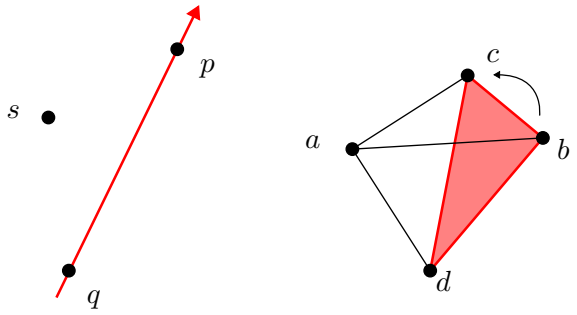

Figure 1.2. Orientation predicate: Point a lies above the plane bcd because it sees the points in counter-clockwise order, $orientation(a, b, c, d) > 0$

0.30000000000000004. We call the distance between two exactly representable floating point numbers *machine epsilon* ε. See figure 1.3 for how this machine epsilon can cause trouble (Example from [**?**]). Because of the imprecision of floating-point arithmetic (FP), logical decisions based on FP should be

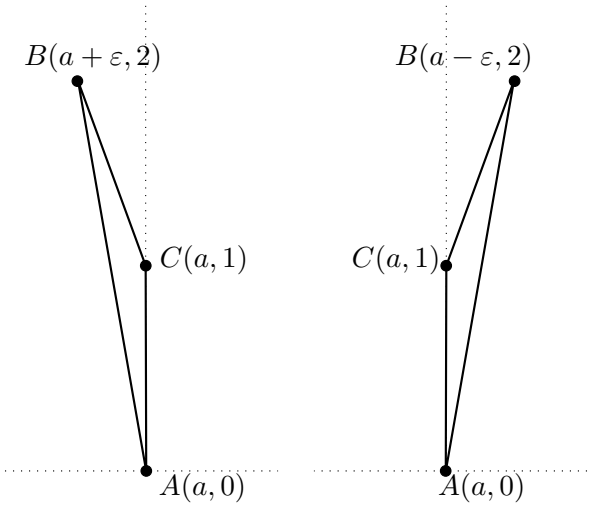

Figure 1.3. The orientation predicate implemented with floating point arithmetic cannot distinguish these two cases. The orientation test calculates $(a + \varepsilon) - a$. If $a = 0$, the orientation test is strictly positive. If $a = 1$, then the result is 0, because during the calculation the machine had to round.

avoided. To fix this issue, we make use of exact predicates that are implemented with arbitrary precision [**?**]. It is assumed that inputs to the orientation test subroutines are exact, and during the calculation, precision is extended as needed such that we can be sure that the result has the correct sign.

1.2 Algorithm *tetFinder*

As described in section 1.1, the straight walk needs different components to work. These components are described in the following subsections. The final algorithm to find all intersected tetrahedra between two points is in Sec. 1.2.5.

1.2.1 Starting tet

To start, the algorithm needs to know in which tet the starting point q is located. With a simple linear nearest neighbor search of point q, we find a good heuristic starting point s. From s, every tet incident to s is checked if it contains q. Most of the times, this method works for locating which tet contains q. In case it fails, an exhaustive search through all the tets of the mesh is done to search which tet contains q. Figure 1.4 shows a constellation where the nearest neighbor approach fails. Checking if a point q is

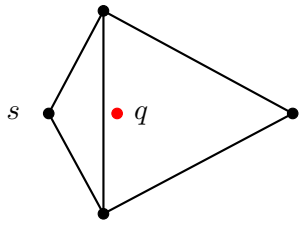

Figure 1.4. The point s is the nearest neighbor to q. However, searching all incident triangles to s, point q cannot be located. A similar example can be constructed in 3D.

within some tet $\mathcal{T} = [a, b, c, d]$ can be done with the orientation predicate, i.e.

$$q \in \mathcal{T} \iff \begin{array}{c} \textit{sameSign}(\textit{orientation}(a,b,c,d),\textit{orientation}(a,b,c,q))\&\& \\ \textit{sameSign}(\textit{orientation}(b,c,d,a),\textit{orientation}(b,c,d,q))\&\& \\ \textit{sameSign}(\textit{orientation}(c,d,a,b),\textit{orientation}(c,d,a,q))\&\& \\ \textit{sameSign}(\textit{orientation}(d,a,b,c),\textit{orientation}(d,a,b,q)) \end{array}$$

By checking for each plane of the tet, we check if the point q is on the same side of the plane (characterised by the same orientation sign) as the remaining vertex.

1.2.2 Valid state and initialization

To walk in the tet mesh, the algorithm needs to be in a state where everything is as expected. In 2D, we define this valid state as

- the line $\vec{qp}$ intersects the edge defined by vertices $\vec{lr}$ and this edge lies between q and p
- vertex l lies on the left of the line defined by $\vec{qp}$
- vertex r lies on the right of the line defined by $\vec{qp}$

If the algorithm starts with these configurations, then in each step the valid configuration will be preserved and all edges intersected are traversed. Analogously in 3D, we define the valid state such that

- the line $\vec{qp}$ intersects the triangle defined by vertices $\triangle uvw$
- vertices of the triangle $\triangle uvw$ are ordered in the way that

$$\textit{orientation}(w, v, q, p) > 0, \textit{orientation}(v, u, q, p) > 0, \textit{orientation}(u, w, q, p) > 0$$

Again, by starting in this configuration, all triangles $\triangle uvw$ traversed get intersected by $\vec{qp}$. Then, we find each intersection point with the ray, add the intersection points to a list with the current tet t and finally use this as input for calculating the rotation coefficient as described in chapter **??**. To get into a valid state, an initialization needs to be performed. In *Walking in a triangulation*[**?**], they assume that q is a vertex in the tet mesh. Thus, their approach of turning around q is not applicable in our case.

We solve this problem with a brute-force approach. First, we find which tet t contains q as described in 1.2.1. From there, we go through the four faces of the tet and check which one gets intersected by the ray $\vec{qp}$. When the face which gets intersected was found, we check the 6 permutations of the three vertices (u, v, w) until the configuration is found, where the orientations are positive.

1.2.3 Robust ray-triangle intersection

When doing the initialization, we check the four faces of the tet to test which face intersects with $\vec{qp}$. This ray-triangle intersection check must be robust, meaning that when the segment $\vec{qp}$ goes exactly through an edge or vertex, it must also be detected. Again, we can use the orientation predicate from before to do this robustly. Let $\triangle uvw$ be a triangle in $\mathbb{R}^3$ and q, p the two points that determine a line segment. If

u, v, w is ordered in the way that *orientation*$(q, u, v, w) > 0$, then the segment determined by $\vec{qp}$ cuts the triangle if and only if [**?**]:

$$\textit{orient}(p, u, q, v) \geq 0 \wedge \textit{orient}(p, w, v, q) \geq 0 \wedge \textit{orient}(p, u, w, q) \geq 0 \tag{1.1}$$

If *orientation*$(q, u, v, w) < 0$, then the orientation tests in Eq. 1.1 must be less or equal to zero. Before checking for intersection with Eq. 1.1, we must ensure that p and q lie on opposite sides of the triangle $\triangle uvw$, i.e.

$$(\textit{orient}(q, u, v, w) < 0 \wedge \textit{orient}(p, u, v, w) > 0) \vee (\textit{orient}(q, u, v, w) > 0 \wedge \textit{orient}(p, u, v, w) < 0) \tag{1.2}$$

TODO: What to do when all 5 points are coplanar There is the possibility that the points q, p, u, v, w are all coplanar (when Eq. 1.2 are all equal zero).

1.2.4 Robustness

The *tetFinder* algorithm is designed to be robust and exact. Exact means no logical decisions based on FP, because this can lead to unexpected and wrong behaviour. Exactness is handled already, because all decisions are made based on the orientation predicate, which is made exact through arbitrary precision arithmetic. Robust in our context means that it is able to handle degenerate cases, for example when the ray goes exactly through an edge or vertex of a tet. The first case to handle is when the starting point q is in multiple tets. Since the tetmesh is not a disjoint union of tets as they share their boundaries, a point can be in multiple tets at once. This can lead to problems in the initialization: If a random tet which contains q is chosen, a valid state must not necessarily be found, see fig. 1.5 To fix this, we add all tets to a list that contain q while searching for the start tet. While initializing, we go through the tets one by one until a valid configuration is found. The cases where the ray lies exactly on a face or goes exactly

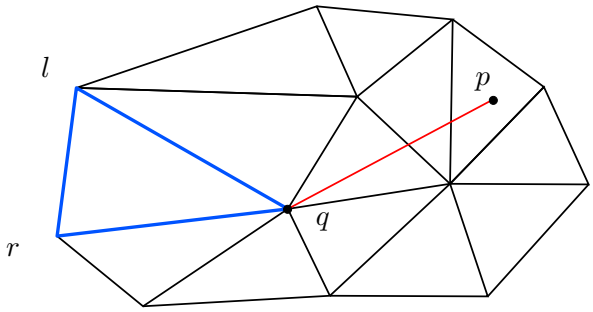

Figure 1.5. As q is exactly on vertex and the chosen triangle (in bold blue) does have an edge that intersects with $\vec{qp}$, it does not matter how l and r are chosen on this triangle, no valid state can be found. The same thing happens in 3D, where no valid configuration for u, v, w can be constructed given a bad tet.

through an edge are all handled similarly. The problem in these cases is that while walking, the next step (reassigning u, v or w to the next vertex) cannot make a valid state, e.g. $\triangle uvw$ do not intersect with $\vec{qp}$ anymore. We detect this by checking if we are still in a valid state and that the end tet does not match with the currently visited tet before the next iteration of the walk. If the walk has not finished yet and we are not in a valid state, we reinitialize from the current tet and use the brute-force approach to search for a valid state again. When found, the walk can continue as normal. Because degenerate cases can lead to invalid states, faces that do not get intersected by $\vec{qp}$ are highlighted by $\triangle uvw$ while walking. Thus, before adding the face $\triangle uvw$ to the list to calculate the intersection points, we quickly check if the face to be added actually intersects with the robust ray-triangle subroutine described above.

1.2.5 Formulation of algorithm

The final algorithm is presented in pseudocode here. The algorithm is described using operations such as:

- `neighbor(t through` uvw`)` returns the tetrahedron sharing face uvw with tetrahedron t
- `s=vertex of` $t, s \neq u, s \neq v, s \neq w$ returns the remaining vertex of a tetrahedron whose other three vertices are known
- `intersection(ray, triangle)` returns the intersection point of the ray with the triangle calculated with FP arithmetic
- `validState(q,p,u,v,w)` checks if the current configuration is valid according to Sec. 1.2.2
- `initialization(t,q,p,u,v,w)` tries to initialize by assigning vertices u, v, w from tet t as described in Sec. 1.2.2. Returns `true` if successful
- `locateTets(q)` returns all tets that contain q

Algorithm 2 contains *tetFinder* in full.

Algorithm 2 tetFinder

```
1: Input (q, p)
2:     STARTS = locateTets(q)
3:     LINESEGMENTS = [ ]
4:     t = NIL // working variables
5:     u, v, w, s = NIL // working variables
6:     for t̂ ∈ STARTS
7:         if (p ∈ t) //p in same tet as q, we are done
8:             return LINESEGMENTS.add([t, q, p])
9:         if (initialization(t, q, p, u, v, w))
10:            t = t̂

11:    // qp intersects triangle uvw
12:    // wvqp, vuqp, uwqp are positively oriented
13:    PREV = q
14:    CURR = intersection(qp, uvw)

15:    LINESEGMENTS.add([t, PREV, CURR])

16:    while orientation(u, w, v, p) > 0 {
17:        if (validState(q,p,u,v,w)) // degenerate cases can lead to invalid states
18:            initialization(t, q, p, u, v, w)

19:        t = neighbor(t through uvw)
20:        s = vertex of t, s ≠ u, s ≠ v, s ≠ w
21:        PREV = CURR
22:        if orientation(u, s, q, p) > 0 // qp does not intersect triangle usw
23:            if orientation(v, s, q, p) > 0 // qp intersects triangle vsw
24:                u = s
25:            else // qp intersects triangle usv
26:                w = s
27:        else // qp does not intersect usv
28:            if orientation(w, s, q, p) > 0 // qp intersects triangle usw
29:                v = s
30:            else // qp intersects triangle vsw
31:                u = s

32:        if (robustRayTriangle(q,p,u,v,w))
33:            CURR = intersection(qp, uvw)
34:            LINESEGMENTS.add([t, PREV, CURR])
35:    } // t contains p
36:    LINESEGMENTS.add([t, PREV, p])
```