Chapter 1

Algorithm for R between two arbitrary points in a mesh

To measure the Dirichlet energy, we need to calculate the rotation coefficient between to 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)
    LINESEGMENTS $\leftarrow$ *tetFinder*(q, p) //returns all tets intersected by the line $\vec{pq}$ with the line segments within them
    $R \leftarrow \mathrm{Id}$
    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 on so called *orientation tests* to determine which triangles we traverse.

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

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

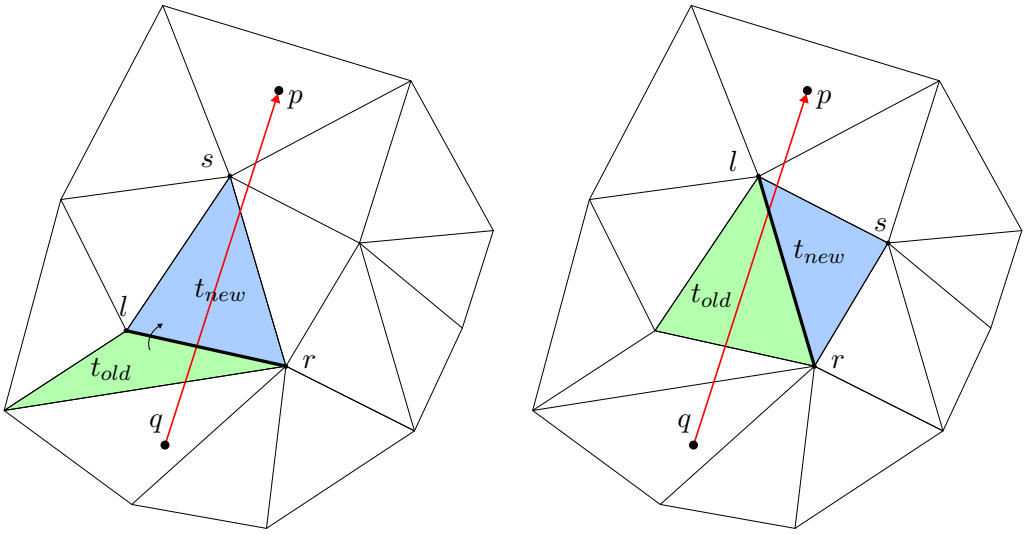

Figure 1.1. Straight walk step

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[**?**]. In most cases, a simple reinitialization is enough when the invalid state is detected.

A note on orientation tests To determine on which side some point s lies relative to two other points q, 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 this case, 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, a famous example of this is $0.1 + 0.2 = 0.30000000000000004$. 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 http://groups.csail.mit.edu/graphics/classes/6.838/S98/meetings/m12/pred/m12.ht
Logical decision based on floating point arithmetic is correct most of the times, but not always. To fix this issue, we make use of exact predicates that are implemented with arbitrary precision [**?**]. Inputs to the orientation test subroutines are assumed to be exact, and we can be sure that the result has the correct sign.

Algorithm 2 tetFinder

```
1: Input (q, p)
2:     t = LocateTet(q)
3:     initialization(t)

4:     // qp intersects triangle uvw
5:     // wvqp, vuqp, uwqp are positively oriented
6:     LINESEGMENTS = [ ]
7:     PREV = q
8:     CURR = intersection(qp, uvw)

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

10:    while orientation(u, w, v, p) > 0 {
11:        t = neighbor(t through uvw)
12:        s = vertex of t, s ≠ u, s ≠ v, s ≠ w
13:        PREV = CURR
14:        if orientation(u, s, q, p) > 0 // qp does not intersect triangle usw
15:            if orientation(v, s, q, p) > 0 // qp intersects triangle vsw
16:                u = s
17:            else // qp intersects triangle usv
18:                w = s
19:        else // qp does not intersect usv
20:            if orientation(w, s, q, p) > 0 // qp intersects triangle usw
21:                v = s
22:            else // qp intersects triangle vsw
23:                u = s
24:        CURR = intersection(qp, uvw)
25:        LINESEGMENTS.add([t, PREV, CURR])
26:    } // t contains p
27:    LINESEGMENTS.add([t, PREV, p])
```

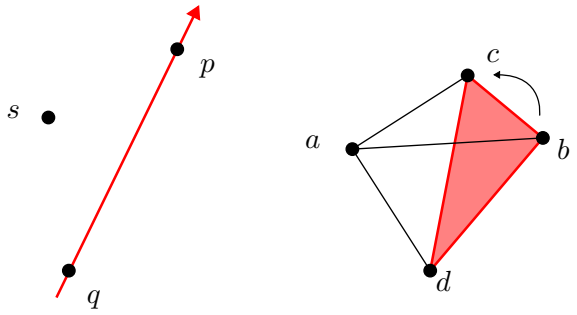

Figure 1.2. Orientation predicate: Point a lies above the plane bcd because it sees the points in counter-clockwise order, $orientation(a, b, c, d) > 0$

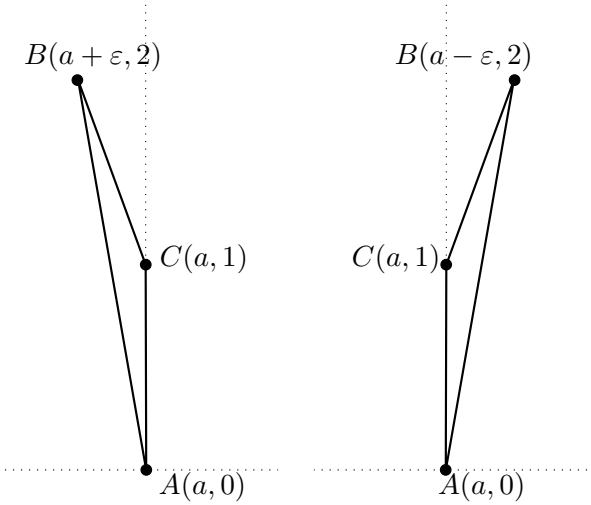

Figure 1.3. The orientation predicate implemented with floating point arithmetic cannot distinguish these two cases. The orientation test calculates $(a + \varepsilon) - a$. If $a = 0$, the orientation test is strictly positive. If $a = 1$, then the result is 0, because during the calculation the machine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

The final algorithm to find all intersected tetrahedra between two points is as follows.
The algorithm uses several subroutines which are described here:

- `neighbor(t through` uvw`)` returns the tetrahedron sharing face uvw with tetrahedron t
- `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
- `initialization()`