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

```
1: Input $(q, p)$
2:     LINESEGMENTS $\leftarrow$ tetFinder($q, p$) //returns all tets intersected by the line $\vec{pq}$ with the line segments within them
3:     $R \leftarrow \mathrm{Id}$
4:     for each SEGMENT in LINESEGMENTS
5:         $R \leftarrow R \cdot$ calcCoeff(SEGMENT)

6: 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 line $\vec{qp}$ is found and the walk repeats. This process is illustrated in Figure 1.1

To determine the on which side a point lies 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

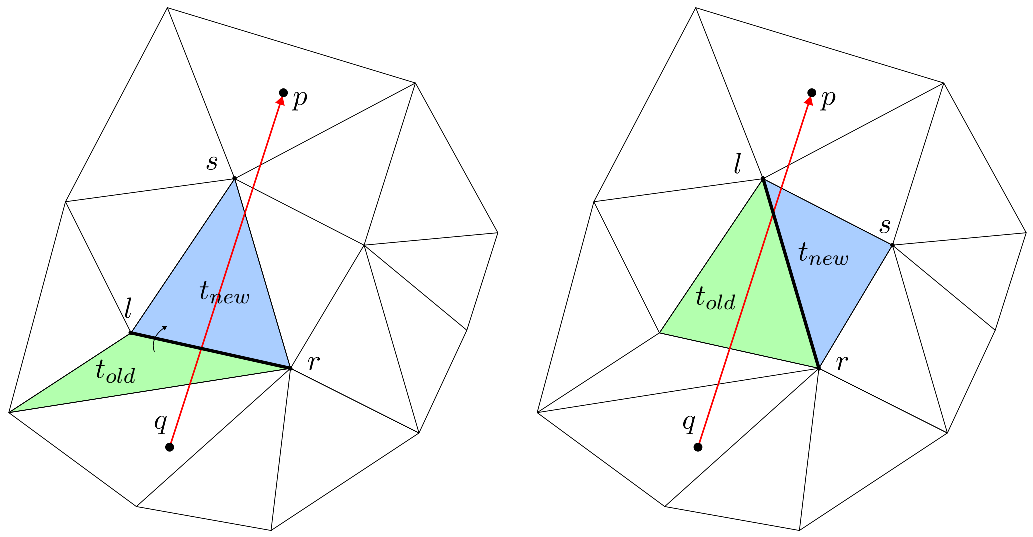

Figure 1.1. Straight walk step

triangle bcd when turning counterclockwise

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

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

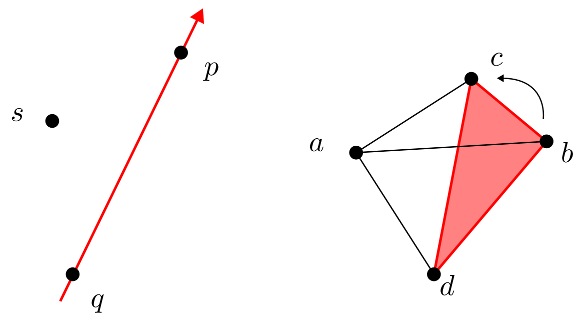

Figure 1.2. Orientation predicate

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.

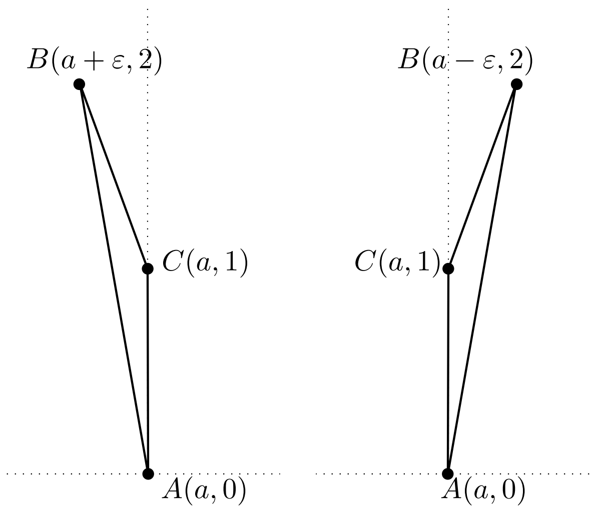

Figure 1.3. The orientation predicate implemented with floating point arithmetic cannot distinguish these two cases. The orientation test calculates $(a+\varepsilon)-a$. If $a=0$, the orientation test is strictly positive. If $a=1$, then the result is 0, because during the calculation the machine had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

Algorithm 2 Byzantine Leader-Based Epoch-Change (process p_i).

```
7: **State**
8:     *lastts* ← 0: most recently started epoch
9:     *nextts* ← 0: timestamp of the next epoch
10:    *newepoch* ← $[\perp]^n$: list of NEWEPOCH messages

11: **upon event** *complain*($p_\ell$) **such that** $p_\ell$ = *leader*(*lastts*) **do**
12:    **if** *nextts* = *lastts* **then**
13:        *nextts* ← *lastts* + 1
14:        send message [NEWEPOCH, *nextts*] to all $p_j \in \mathcal{P}$

15: **upon** receiving a message [NEWEPOCH, *ts*] from $p_j$ **such that** *ts* = *lastts* + 1 **do**
16:    *newepoch*[$j$] ← NEWEPOCH

17: **upon exists** *ts* **such that** $\{p_j \in \mathcal{P} |$ *newepoch*[$j$] = *ts*$\} \in \mathcal{K}_i$ **and** *nextts* = *lastts* **do**
18:    *nextts* ← *lastts* + 1
19:    send message [NEWEPOCH, *nextts*] to all $p_j \in \mathcal{P}$

20: **upon exists** *ts* **such that** $\{p_j \in \mathcal{P} |$ *newepoch*[$j$] = *ts*$\} \in \mathcal{Q}_i$ **and** *nextts* > *lastts* **do**
21:    *lastts* ← *nextts*
22:    *newepoch* ← $[\perp]^n$
23:    **output** *startepoch*(*lastts*, *leader*(*lastts*))
```