Building Better Intrinsic Triangulations

Sam Ruggerio (samuelr6), Alex Zhang (alexmz2), Flora Zheng (floraz2)

May 2023

1 Introduction

3D meshes are used in a wide range of domains from augmented reality to computer vision, gaming, animation, and simulation. A surface mesh encodes the 3D geometry of a surface. Most formats used in practice describe a surface extrinsically, using points in 3D space and describing the connectivity of edges and faces. This is fine for most applications. However, an extrinsic description of a surface is only one way to represent that surface. Additionally, the quality of the mesh can have poorly shaped triangles (long, skinny, etc.) which affect the accuracy of many geometric processing algorithms. Intrinsic meshes represent the surface with edges that go along the surface, with no direct reference to how it sits in real space.

We can refine meshes with intrinsic triangulations to have "better" triangles, acute angles with similar edge lengths, which improves the performance and stability of many geometric algorithms. This process is called Delaunay Refinement, and is the major benefit to using intrinsic triangulations to represent surfaces. One can use these geometric algorithms, almost directly, by a change to use edge lengths instead of vertices. This has been applied to Steiner tree approximation, finite element analysis, and more.

There has been work in representing intrinsic triangulations with efficient data structures. In this report, we will describe what these structures are and the limitations of their implementation. We then move to what we are trying to improve in the data structure, and our various attempts. Finally, we landed on a change that should improve numeric stability of one of the structures by avoiding calls to transcendental functions.

2 Notation

We are often dealing with multiple types of meshes/triangulations $T = (V, E, F)$. To keep things in order, we denote the input triangulation as T^0, which represents our extrinsic triangulation. The intrinsic mesh is denoted T^1. In most cases, we are only ever adding vertices to T_1, as modifying the mesh of T_0 would affect the representation of the surface. In other words, $V_0 \subseteq V_1$.

3 Related Work

3.1 Signposts

Sharp et al. [3] present a data structure to represent intrinsic triangulations. This data structure stores an intrinsic mesh which has improved element quality and mesh size, which allows complex algorithms from scientific computing and computational geometry to run on low quality input meshes. In a signpost data structure, each vertex has a signpost that stores the distance and direction of each vertex from adjacent vertices.

The signpost data structure consists of $T^1 = (V, E, F)$, edge lengths $\ell : E \to \mathbb{R}_{>0}$, and angles $\phi : H \to [0, 2\pi)$ where H is the set of half-edges. Each angle ϕ_{ij} has the direction of the halfedge from vertex i to vertex j using the polar coordinate system for vertex i, based off of a tangent space respective to an arbitrary halfedge leaving i.

The signpost data structure supports two atomic operations: signpost updates and tracing queries. The signpost update operation maintains the angle stored at each half edge leaving a vertex i. For example, say

we need to update the angle of the halfedge ik, which lies in $\triangle ijk$. We first take the angle of ij, ϕ_{ij}, which is known, compute $\angle jik$ via the edge lengths of the triangle, then add it to ϕ_{ij}.

A tracing query takes an input point p, a unit direction u, and a distance s and outputs the corresponding triangle ijk and barycentric coordinates representing q. This is done by effectively raycasting u until it intersects a halfedge, then changing coordinate systems to go into the next triangle. This repeats until the raycast no longer leaves the triangle. The final barycentric coordinates are then computed.

A vertex update can be implemented with the signpost updates and tracing queries, thus giving us the ability to insert and change the connectivity of T^1. Many other operations can be implemented with the atomic operations, including edge flips and vertex inserts.

Finally, there is a bijection between points in the intrinsic representation and points in the extrinsic representation. The signpost data structure also supports queries of the following form: given a point in the intrinsic representation, locate the corresponding point in the extrinsic representation and vice versa.

3.2 Integer Coordinates

Gillespie et al. [1] present a way to construct intrinsic meshes of Euclidean polyhedra by encoding surfaces with integer coordinates. Consider T^1 overlaid with T^0. We let n_{ij} be the normal coordinate of an edge $ij \in E_1$ and we let c^{ij} be the number of times it crosses edges in E_0. If an edge is shared between E_0 and E_1, we set the normal coordinate to -1. This scheme allows us to store important intrinsic information with just integers. We can recover the overlay and perform vertex insertions, flips, and tracing queries by maintaining and utilizing the normal coordinates. This has the same end goal of improving and refining an input mesh, so that various geometric algorithms can have a higher quality input from low quality meshes.

In an intrinsic triangulation, curves start and end at vertices and can travel parallel to mesh edges. For edge ij, a positive value of n_{ij} indicates the number of crossings, while a negative value means the edge has k parallel curves. We can then define the number of crossings as $n_{ij}^+ = \max(n_{ij}, 0)$ and the number of parallel curves as $n_{ij}^- = -\min(n_{ij}, 0)$.

For each vertex, the number of edges exiting the vertex and the number of edges crossing the vertex can be computed with the normal coordinates. For triangle ijk, let e_k^{ij} denote the number of edges exiting vertex k. Let the corner coordinate c_k^{ij} denote the number of edges crossing vertex k.

$$e_k^{ij} = \max(0, n_{ij}^+ - n_{jk}^+ - n_{ki}^+)$$
$$c_k^{ij} = \frac{1}{2}(\max(0, n_{jk}^+ + n_{ki}^+ - n_{ij}^+) - e_i^{jk} - e_j^{ki}$$

The normal coordinate is then

$$n_{ij} = c_j^{ik} + e_k^{ij} + c_i^{jk}$$

The corner coordinates are useful for following the curve. A curve in a triangle can travel left, intersect the vertex on the other side, or travel right. c_j^{ik} measures the number of curves that travel left, while c_i^{kj} measures the number of curves that travel right.

The tracing returns the edges that the curve crosses. However, if the curves are geodesic, then the curve can be traced by unfolding the triangles of the intrinsic mesh in the plane. We then draw a straight line through the endpoints formed by the intersection points of the curve and the triangles. For low-quality meshes, error from computing with floating point numbers can be introduced in the step.

In addition, normal coordinates can encode a set of curves. The union of two disjoint curves is the sum of the normal coordinates. The intrinsic triangulation is determined by the number of times each curve intersects edges in the extrinsic mesh. Normal coordinates can be calculated for each edge of the extrinsic triangulation or for each edge of the intrinsic triangulation. The approach taken is to compute normal coordinates for each edge of the intrinsic triangulation.

Roundabouts augment the information represented by normal coordinates, since normal coordinates do not provide enough information to determine the correspondence between the two triangulations. Given a curve, normal coordinates do not identify the corresponding edge - the endpoints of an edge are not a unique identifier for the edge. Let u be a vertex of both the extrinsic and intrinsic mesh. For an intrinsic halfedge $\overrightarrow{uv}$, the roundabout represents the first extrinsic halfedge $\overrightarrow{uw}$ after the intrinsic halfedge. Suppose the order of halfedges in the exterior triangulation with endpoint u that are directed away from vertex u is given in

counterclockwise order. A roundabout is $r_{\overrightarrow{uv}} \in \mathbb{Z}_{\geq 0}$. Roundabouts determine the order of edges swinging outward from a vertex.

Finally, a polygon mesh between two triangulations is called the common subdivision. The common subdivision allows intrinsic data at vertices and vertex positions for the extrinsic representation to be computed. The data structure supports edge flips.

4 Problem Statement

Sharp et al. [3] and Gillespie et al. [1] present extremely performant and robust implementations for intrinsic mesh processing. However, each implementation has their own limitations. For the signpost data structure, atomic operations are extremely fast, but the ray casting and angle updates involved leads to many calls to transcendental functions (such as arccos, cos, tan). This results in some cases where the data structure fails to properly trace out intrinsic triangles and get an extrinsic correspondence back due to the numerical instability of these transcendental functions.

On the other hand, the integer coordinate data structure avoids numerical instability, since everything is operating over integers. However, while tracing queries are still fast and now numerically stable, vertex insertions are not. This is because the data structure needs to trace the extrinsic mesh to determine where the inserted point will lie within a triangle, so that the new normal coordinates of each outgoing edge is determined. At worst, you will have the case where $O(n)$ edges cross $O(n)$ edges, resulting in $O(n^2)$ intersections. Gillespie et al. [1] mention that storing these intersections is an optimization, and one can implement the data structure where they are computed on the fly, but do not explore this any further.

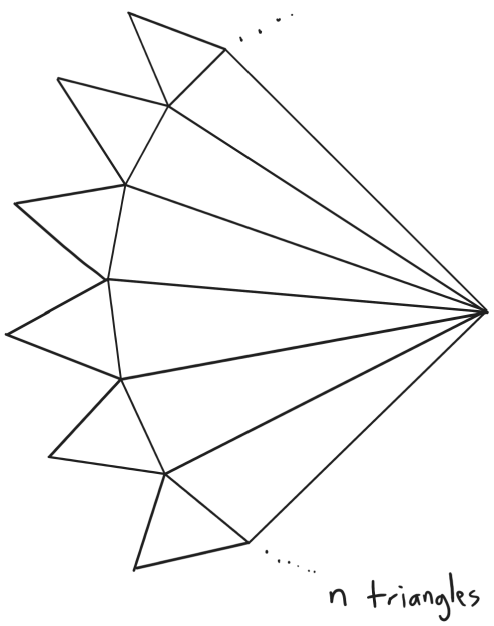
So we're left with a trade off: we can represent an intrinsic triangulation with fast vertex insertions, but unstable tracing queries. On the other hand, we can represent an intrinsic triangulation with slow vertex insertions, but stable tracing queries. Our goal is then to describe a data structure which ideally has fast vertex insertions and stable tracing queries. Gillespie et al. [1] propose that a hybrid data structure might be a possible solution. We will first look into having quicker access to normal coordinate intersections, which we've named mile markers, in order to try and avoid this hybridization. Next, we'll discuss our attempts at building a hybridized data structure. Finally, we'll discuss our improvement to the signpost data structure that uses slopes instead of angles in order to completely avoid the transcendental functions and maintain numerical stability.

5 Mile Markers

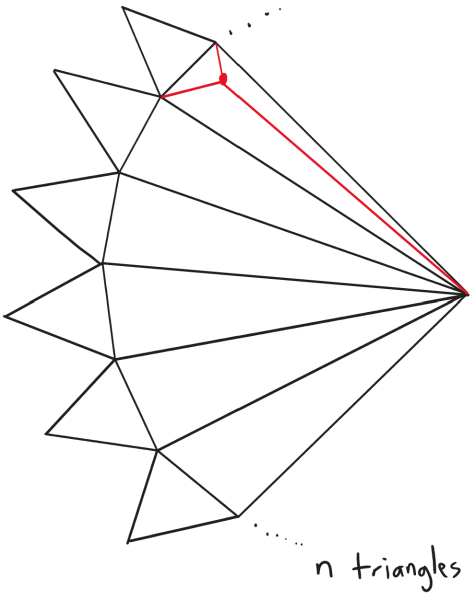
We propose a new item to keep track of in our data structure: mile markers. Instead of storing normal coordinates n_{ij} for each edge $ij \in E^1$, where the normal coordinate corresponds to how many edges in E^0 crosses over ij, we instead store mile markers. Each point of intersection between an extrinsic and an intrinsic edge gets one mile marker, stored as a property of the intrinsic edge ij. The mile marker stores the distance from vertex i to the intersection point, as well as which extrinsic edge is involved.

Intuitively, it appears that these mile markers would help speed the atomic operations up. Inserting vertices into triangle ijk is simple, since the data structure knows exactly where the extrinsic edges come into the triangle, so it can calculate the normal coordinates and new mile markers in linear time with respect to the number of mile markers involved. Edge flipping entails deleting an edge and adding a new one - but with all the information about the extrinsic edges in the quadrilateral already stored, finding the new normal coordinates and mile markers is also a task that can be done in linear time with respect to the number of mile markers involved.

Therefore, it appears that this would speed up the slow vertex insertions, while keeping the edge flips at the same, linear time complexity. The cost would be the extra storage needed for these mile markers. The question then becomes: how many mile markers will be created globally? Every intersection between an intrinsic edge and an extrinsic edge requires the creation of a mile marker, so the overall complexity of these vertex insertions and edge flips also depends on the number of mile markers. We now show that in the worst-case, the number of mile markers created is lower-bounded at $O(n^2)$, where n is the number of extrinsic edges.



(a) Initial extrinsic triangulation



(b) Insert an intrinsic vertex at an arbitrary point

Figure 1

Here is a proof by picture, i.e. a counterexample. For all figures in this section, the black edges are the extrinsic edges, while the red edges are the intrinsic edges. We start with an extrinsic triangulation, shown in Figure 1a, but flattened out so we can easily draw on the triangulation. This happens to be a flattened regular pyramid with a base of n sides, giving $n+1$ vertices, $3n$ edges, and $2n$ triangles in the extrinsic triangulation.

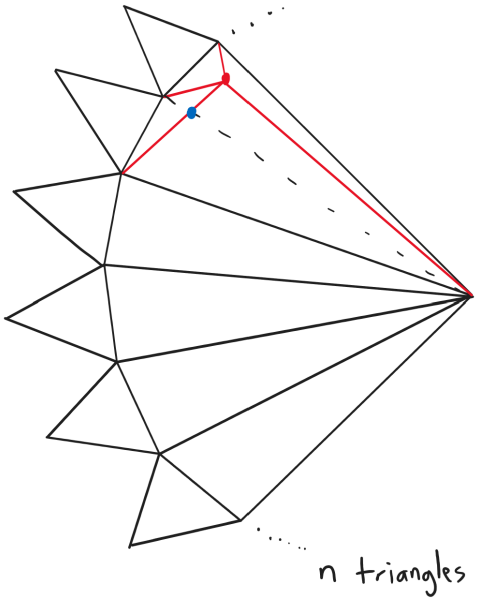
We then insert an intrinsic vertex along with the accompanying edges, all shown in Figure 1b in red. We can then perform a series of edge flips as shown in Figure 2a and 2b. These edge flips will introduce intrinsic edges that cross extrinsic edges; we draw these mile markers in blue in the figures. The first edge flip introduces one mile marker, since the newly introduced intrinsic edge only crosses one extrinsic edge. The second edge flip introduces two mile markers, one for each extrinsic edge it crosses. The third edge flip introduces three, and so-on.

Therefore, the total number of mile markers equals $1+2+\cdots+(n-1) = O(n^2)$ lower-bound, since given n triangles, we will perform the edge flip $n-1$ times. Note that the flattened regular pyramid is not the only shape that has this feature; the edges corresponding to the perimeter of the base of the pyramid can be straightened out, while still keeping the structure of n triangles stacked on top of each other, so that the construction can support any integer n and ignore angle considerations (the resulting diagram will no longer be a flattened regular pyramid, but that is irrelevant). Therefore, since the number of mile markers created is lower-bounded by $O(n^2)$, there can be no actual speedup by storing these mile markers.

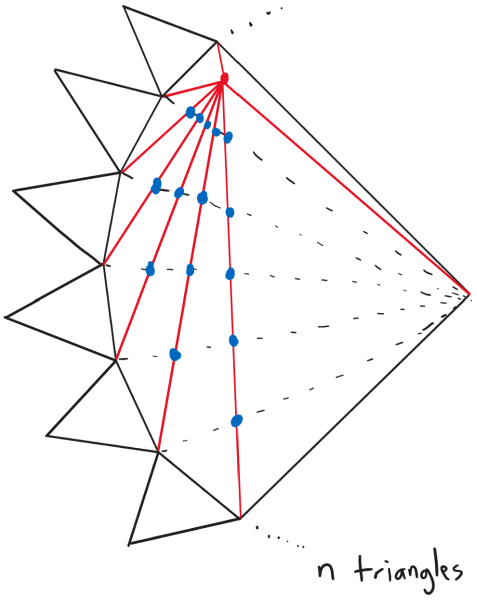
5.1 Lazy Evaluation and/or Caching

Gillespie et al. [1] recognized that vertex insertion and the calculation of normal coordinates was very inefficient, since they needed to compute the geometric crossings of all extrinsic curves passing through the triangle in question. They proposed some optimizations, namely lazy evaluation or caching, as a quick sidenote, but did not analyze them. In this section, we show that neither of these optimizations will actually help in speeding up vertex insertions.

The lazy evaluation idea says that the data structure should only compute the geometric crossings of all extrinsic curves passing through the triangle once the calculations are actually needed. However, when looking at the counterexample shown in Section 5, it can be seen that the second edge flip will require the



(a) First edge flip. The intrinsic edges are drawn in red, while the mile marker location is drawn in blue.



(b) Subsequent edge flips.

Figure 2

geometric crossings that were generated by the first edge flip, since the quadrilateral in which the edge flip is taking place includes the intrinsic edge generated by the first edge flip. The third edge flip will require the geometric crossings that were generated by the second edge flip as well. Therefore, the total number of geometric crossings that need to be computed is now $1 + 2 + \cdots + (n-2)$ instead, since the last triangle's geometric crossings do not need to be computed. However, this is still lower-bounded by $O(n^2)$ calculations. No speed-up here.

The caching idea involves storing the computed geometric crossings on the intrinsic edge once they have been calculated. Unfortunately, the entities stored in this cache are exactly the mile markers we discussed earlier, which means the size of the caches will reach $O(n^2)$ worst-case. Still no speed-up here.

In fact, the counterexample in Section 5 allows us to conclude that any data structure using normal coordinates in this fashion will be forced to find the new normal coordinates using some method other than computing them one by one. Normal coordinates here represent how many times extrinsic edges crosses the specified intrinsic edge, which means the normal coordinate stored on the intrinsic edge is exactly equal to the number of mile markers stored on that edge. The total number of mile markers is then equal to the sum of all normal coordinates in the whole data structure, which we showed to be $O(n^2)$ worst case.

6 Hybridization

Since optimizing the provided data structure proved to be difficult, we then attempted to combine the data structures presented by Sharp and Gillespie in some meaningful way. The idea would be to map some atomic operation between the partial data structures and maintain correspondence without accumulating errors or misrepresenting any information within the data structure. The normal coordinate data structure has the most bookkeeping necessary, so our intuition led to keeping it as the base of our new data structure.

Now, we treat inserting vertices much like signposts. We insert a new vertex and three edges into T^1. This subdivides a triangle into three regions. We can compute the new edge lengths based on the barycentric coordinates of the inserted vertex, as well as compute the direction of the new edges. To compute the normal coordinates of these new edges, we only need to perform three tracing queries via the signpost method. Each

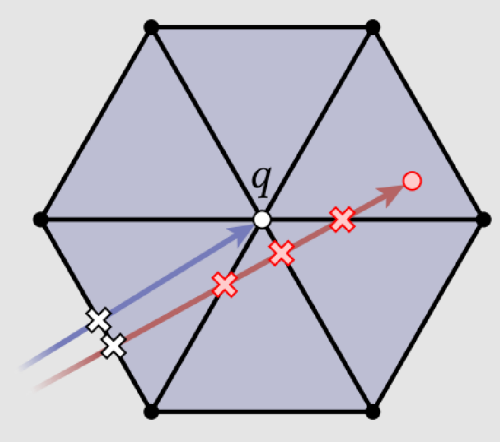


Figure 3: A trace entering the star simplex of q, from [3]

tracing query will return a list S of barycentric coordinates along the edge of crossings from T_0. We can set the normal coordinate for that edge to $|S|$.

Like the original implementation, we can still insert vertices on an edge, subdividing two triangles into four. We can still update normal coordinates with three tracing queries. This is accomplished by tracing along the halfedge from the inserted vertex to one of the edge's endpoints. Then we can set the normal coordinate of the opposite edge to the original edge's normal coordinated subtracted by $|S|$. Once the normal coordinates are determined, we treat the new connectivity as a normal coordinate data structure, and all further tracing queries are handled through normal coordinates. It should be noted that this does not completely remove the numerical instability of signpost tracing queries, but may be more robust than the general implementation.

The reason for this improvement in stability is noted in Appendix A of Sharp et al. [3]. If we know the target point q (which we have a direct reference to, since we're tracing along an edge and stopping at the end of the edge), then we can perform a few operations to counteract instability. In particular, when we trace towards a point q, the star of the simplex σ that contains q must be crossed at the end of the trace. Any further crossings are erroneous, and can be removed. See 3 for an example, erroneous crossings are marked in red, and the correct trace is marked in blue. Additional improvements can be applied to target points that are not shared with V_0, in which case we can do a local search of nearby triangles, or snap the query line to a nearby edge.

7 Slopes Over Angles

However, the signpost data structure uses angles and transcendental functions, which can cause numerical instability. We now propose a new data structure that uses linearposts, as opposed to signposts in order to avoid this problem. But first, some coordinate systems.

7.1 Barycentric Coordinate System

The algorithms in later sections use the barycentric coordinate system heavily. Below are the operations that we will need and their formulae:

- Conversion from barycentric to Cartesian: Given the Cartesian coordinates of the triangle's vertices (x_1, y_1), (x_2, y_2), and (x_3, y_3) and a barycentric point $p = (p_1, p_2, p_3)$, we can convert this point to its Cartesian form (x_p, y_p) using the formula

$$\begin{aligned} x_p &= p_1 x_1 + p_2 x_2 + p_3 x_3 \\ y_p &= p_1 y_1 + p_2 y_2 + p_3 y_3 \end{aligned} \tag{1}$$

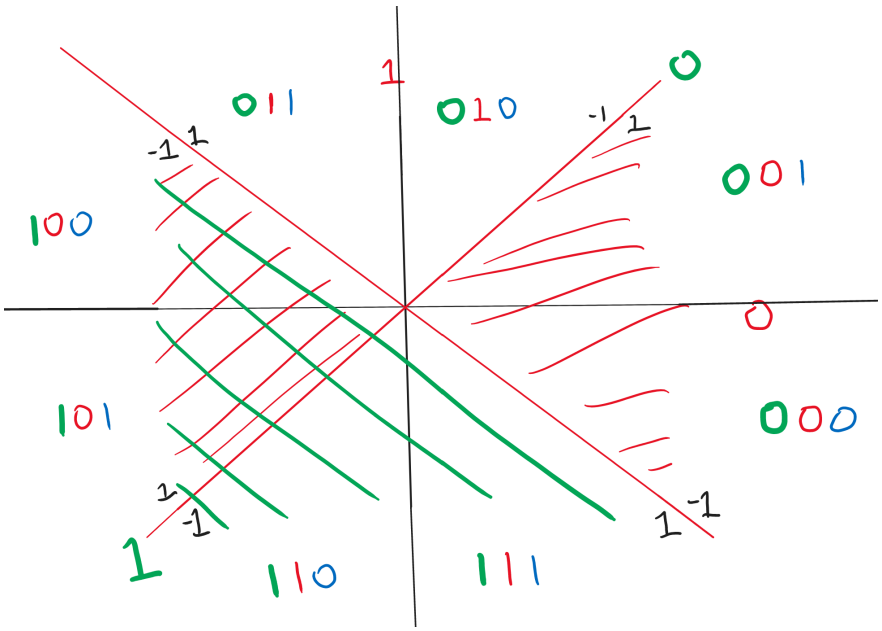


Figure 4: Slope with reciprocal-bit coordinate system. The red numbers are the reciprocal-bit r, the green numbers are the side bit s, the blue numbers are the sign of the slope, while the black numbers are the corresponding slopes at those locations. The red-shaded region is the region where the reciprocal bit is 0, while the region that isn't red is where the reciprocal bit is 1. The green-shaded region is the region where the sign bit is 1, while the region that isn't green is where the sign bit is 0.

- Conversion from Cartesian to barycentric: Given the Cartesian coordinates of the triangle's vertices (x_1, y_1), (x_2, y_2), and (x_3, y_3) and a Cartesian point $p = (x_p, y_p)$, we can convert this point to its barycentric form (p_1, p_2, p_3) using the formula

$$\begin{aligned} p_1 &= \frac{(y_2 - y_3)(x_p - x_3) + (x_3 - x_2)(y_p - y_3)}{(y_2 - y_3)(x_1 - x_3) + (x_3 - x_2)(y_1 - y_3)} \\ p_2 &= \frac{(y_3 - y_1)(x_p - x_3) + (x_1 - x_3)(y_p - y_3)}{(y_2 - y_3)(x_1 - x_3) + (x_3 - x_2)(y_1 - y_3)} \\ p_3 &= 1 - p_1 - p_2 \end{aligned} \tag{2}$$

- Distance: The distance between two barycentric coordinates on a triangle is provided by Schindler and Chen [2]. For the sake of completion, we include it here. Given the three edge lengths for triangle ijk and barycentric points p and q for two points on this triangle, we can find the distance d between p and q using Equation 3.

$$u := p - q$$

$$d = \sqrt{-(\ell_{ij}^2 u_1 u_2 + \ell_{jk}^2 u_2 u_3 + \ell_{ki}^2 u_3 u_1)} \tag{3}$$

7.2 Slope Coordinate System

Figure 4 show the coordinate system that we will be using for slopes. Each slope is represented by a real number between -1 and 1, which we will denote m, and two section-bits s and r, where s is the side bit and r is the reciprocal bit. Notice that this is essentially four coordinate systems stitched together, with the two

bits s and r determining which coordinate system we are actually using. The slope itself is then clamped to be between -1 and 1, which is good for numerical stability reasons - the slope will never explode!

The xy plane is then divided into 8 sections, one for each combination of s, r, and b, where b is the sign of the slope m. We set $b = 1$ if the slope is negative, 0 else). If the bits are arranged in the order srb and interpreted as a binary number $[srb]$ (square brackets means interpret bits as an integer), we find that each section is the next binary number from the section right before it, if we traverse in a counterclockwise fashion. For the remainder of this report, this slope/reciprocal-bit coordinate system will simply be referred to as the Slope Coordinate System (SCS).

For the remainder of this report, we will often say slope of ik with respect to ij, where ik and ij are two line segments joined together at one of the ends at point i. This means that the slope information (m, s, r) of ik is calculated by setting ij to be the x-axis. We often need this because we would like this coordinate system to remain local with respect to individual edges to save computation time.

There are a few operations that we would like to support in this coordinate system. We enumerate them here.

- Conversion to real slope: the name is self-explanatory. We return m', where:

$$m' = \begin{cases} m & \text{if } [sr] = 00 \\ -\frac{1}{m} & \text{if } [sr] = 01 \\ -m & \text{if } [sr] = 10 \\ \frac{1}{m} & \text{if } [sr] = 11 \end{cases} \tag{4}$$

(5)

- Conversion from Cartesian to slope: Given the Cartesian coordinates of a point (x, y), we would like to return the slope in the format (m, s, r). We can do this as follows:

$$\begin{aligned} s &= (|x| \leq |y|) \\ r &= (y \leq -x) \\ m &= \begin{cases} \frac{y}{x} & \text{if } r = 0 \\ -\frac{x}{y} & \text{if } r = 1 \end{cases} \end{aligned} \tag{6}$$

- Conversion from slope to unit Cartesian: Given the slope in (m, s, r) form, we would like to return the Cartesian coordinates of the point (x, y) if we were to travel a distance of exactly 1 from the origin. We can do this by first solving the following system of equations to get x' and y', then adjusting these values to get the final value of x and y based on the s and r bits.

  The systems of equations to solve are:

$$\begin{aligned} x'^2 + y'^2 &= 1 \\ \frac{y'}{x'} &= m \end{aligned}$$

$$x' = \sqrt{\frac{1}{1+m^2}} \tag{7}$$

$$y' = \sqrt{\frac{m^2}{1+m^2}} \tag{8}$$

$$x = \begin{cases} x' & \text{if } [sr] = 00 \\ -y' & \text{if } [sr] = 01 \\ -x' & \text{if } [sr] = 10 \\ y' & \text{if } [sr] = 11 \end{cases} \tag{9}$$

$$y = \begin{cases} y' & \text{if } [sr] = 00 \\ x' & \text{if } [sr] = 01 \\ -y' & \text{if } [sr] = 10 \\ -x' & \text{if } [sr] = 11 \end{cases} \tag{10}$$

- Conversion from slope to Cartesian with length: Given the slope in (m, s, r) form, we would like to return the Cartesian coordinates of the point (x, y) if we were to travel a distance of exactly ℓ from the origin, where ℓ is given. To do this, we simply run the conversion from slope to unit Cartesian, then multiply the result by ℓ.

- Slope chaining: We are given the Cartesian coordinate system with the origin labelled i and three other points j, k, l, ordered counterclockwise from the x-axis. We are also given the slope of ik with respect to ij and the slope of il with respect to ik. We would like to find the slope of il with respect to ij. The key here is to find the basis vectors $\vec{x}_{ik}$ and $\vec{y}_{ik}$ of ik with respect to ij, so that the slope can easily translate between these coordinate systems.

  We start by converting the slope of il with respect to ik, which we denote (m_{il}, s_{il}, r_{il}), into unit Cartesian form (x_{il}, y_{il}). We also convert the slope of ik with respect to ij, which we denote (m_{ik}, s_{ik}, r_{ik}), into unit Cartesian form (x_{ik}, y_{ik}). But by the definition of "with respect to ik", (x_{ik}, y_{ik}) must exactly be the basis vector $\vec{x}_{ik}$ with respect to ij.

  The basis vector $\vec{y}_{ik}$ must be the unit vector perpendicular to $\vec{x}_{ik}$. There are likely ways to do this while remaining in two dimensions, but the cross product gives us exactly what we want and is much easier conceptually. In three-dimensional space, we know $\hat{x} \times -\hat{z} = \hat{y}$. Therefore, we extend $\vec{x}_{ik}$ with a 0 to make it three-dimensional, then cross it with $[0, 0, -1]$. The resulting vector's z-component will be 0, so we drop it. This is then $\vec{y}_{ik}$.

  We now have the basis vectors of ik with respect to ij. The coordinates (x_{il}, y_{il}) are in the ik basis vector. Therefore, our final answer, denoted (x_f, y_f) is then $x_{il}\vec{x}_{ik} + y_{il}\vec{y}_{ik}$. We can then convert this back into slope form and return it. Writing this out as matrices, this corresponds to

  $$\begin{bmatrix} x_f \\ y_f \end{bmatrix} = \begin{bmatrix} \vec{x}_{ik} & \vec{y}_{ik} \end{bmatrix} \begin{bmatrix} x_{il} \\ y_{il} \end{bmatrix}$$

  The column matrix is actually a 2×2 orthonormal matrix representing the basis vectors of ik with respect to ij. We will denote this matrix as T_{ij}^{ik}, since it transforms vectors from the ik basis (upper index) into the ij basis (lower index).

- Slope difference: We are given the Cartesian coordinate system with the origin labelled i and three other points j, k, l, ordered counterclockwise from the x-axis. We are also given the slope of ik with respect to ij and the slope of il with respect to ij. We would like to find the slope of il with respect to ik.

  Once again, the key here is to find the basis vectors $\vec{x}_{ik}$ and $\vec{y}_{ik}$ of ik with respect to ij, so that the slope can easily translate between these coordinate systems. To do this, we can simply repeat the steps described in the slope chaining algorithm to obtain T_{ij}^{ik}. We then have, analogous to the previous section:

  $$\begin{bmatrix} x_{il} \\ y_{il} \end{bmatrix} = T_{ij}^{ik} \begin{bmatrix} x_f \\ y_f \end{bmatrix}$$

where the indices have swapped, since we now know slope il with respect to ij and want to find slope il with respect to ik instead. This equation implies we need to invert T_{ij}^{ik}, although since T_{ij}^{ik} is orthonormal, we can just take the transpose. We can interpret the resulting matrix as T_{ik}^{ij}, since it transforms vectors from the ij basis into the ik basis.

$$\begin{bmatrix} x_f \\ y_f \end{bmatrix} = T_{ik}^{ij} \begin{bmatrix} x_{il} \\ y_{il} \end{bmatrix}$$

Once we have the Cartesian coordinates, we can convert back to SCS as usual.

An astute reader might notice that all we are doing is effectively coordinate-transforming between rotated Cartesian systems, and that there are matrix/inverse matrix operations as a function of angle θ that represent rotations that can do all this for us. The reason we explicitly avoid building these matrices in this manner is that to build them, we would have to find the angle, then use transcendental functions to find the matrix and evaluate the rotation. This is exactly what we want to avoid! Thus, we build the matrices a different way instead.

7.2.1 Other Operations

There were some other operations in this coordinate system that we found that we could do, but are not necessary for the remainder of the paper. We include them here for completeness, but will not provide pseudocode for them. They are:

- Overflow m: Suppose we have a coordinate (m, s, r), but $m > 1$. This case can occur if we need addition so we describe it here.
  - $(s, r) = (0, 0)$: $(m, s, r) \longrightarrow \left(-\frac{1}{m}, 0, 1\right)$
  - $(s, r) = (0, 1)$: $(m, s, r) \longrightarrow \left(-\frac{1}{m}, 1, 0\right)$
  - $(s, r) = (1, 0)$: $(m, s, r) \longrightarrow \left(-\frac{1}{m}, 1, 1\right)$
  - $(s, r) = (1, 1)$: $(m, s, r) \longrightarrow \left(-\frac{1}{m}, 0, 0\right)$
- Underflow m: Suppose we have a coordinate (m, s, r), but $m < -1$. This case can occur if we need addition so we describe it here.
  - $(s, r) = (0, 0)$: $(m, s, r) \longrightarrow \left(-\frac{1}{m}, 1, 1\right)$
  - $(s, r) = (1, 1)$: $(m, s, r) \longrightarrow \left(-\frac{1}{m}, 1, 0\right)$
  - $(s, r) = (1, 0)$: $(m, s, r) \longrightarrow \left(-\frac{1}{m}, 0, 1\right)$
  - $(s, r) = (0, 1)$: $(m, s, r) \longrightarrow \left(-\frac{1}{m}, 0, 0\right)$
- Comparison: Suppose we have two coordinates (m_1, s_1, r_1) and (m_2, s_2, r_2), and we wish to evaluate the Boolean expression $(m_1, s_1, r_1) < (m_2, s_2, r_2)$. In this case, we will define our comparison to follow the logic as if these were actually angles.

  Since angles only need to range from 0 to 2π, we would like the section with srb code 001 to be the smallest, while the section with srb code 000 to be the biggest. Therefore, we take the srb codes of the two coordinates, interpret them as integers between 0 and 7, and subtract 1 from them (with 0 mapping to 7). We then use the following rule for comparisons:
  - If $([s_1 r_1 b_1]\%8) != ([s_2, r_2, b_2]\%8)$, return $([s_1 r_1 b_1]\%8) < ([s_2, r_2, b_2]\%8)$. Otherwise, return $m_1 < m_2$.

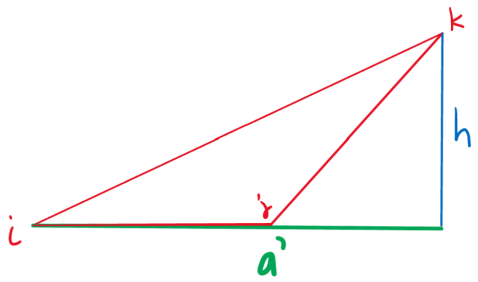


Figure 5: Lengths calculated to get slope of ik with respect to ij. Triangle ijk is drawn in red.

7.3 Linearpost Data Structure

The linearpost data structure can still be built on top of any standard mesh data structure, although we will continue to use a halfedge mesh to be consistent with Gillespie et al. [1] Starting with $M = (V, E, F)$, our linearpost data structure adds the following additional pieces of data:

1. Positive edge length $\ell \ : \ E \longrightarrow \mathbb{R} > 0$.
2. Four tuples (m, s, r) stored at each halfedge, such that $m \in \mathbb{R}, -1 \leq m \leq 1$, $s \in \{0, 1\}$, and $r \in \{0, 1\}$, representing slopes in SCS.

Each edge ij is connected to two triangles, while the total number of edges across these two triangles is 5. To make later calculations convenient and not have to worry about recalculating these tuples, we choose to store the slope of all of these edges with respect to ij on the ij halfedge.

For each vertex i inserted into the intrinsic mesh, we still store its location via a pointer to the extrinsic edge or triangle it belongs two, and its barycentric coordinates b_i within that element.

7.4 Atomic Operations

7.4.1 Triangle Update

Often, we will have situations where we know all three edge lengths of the triangle. We then need to compute the slopes for all three vertices. Let the triangle be denoted ijk, with edge lengths ℓ_{ij}, ℓ_{jk}, and ℓ_{ki} known. We need to find the slope of ik with respect to ij, jk with respect to ji, kj with respect to ki, and all the opposite versions.

Without loss of generality, let us find the slope of ik with respect to ij. The area of triangle ijk can be calculated using Heron's formula:

$$c := \frac{\ell_{ij} + \ell_{jk} + \ell_{ki}}{2} \tag{11}$$

$$A = \sqrt{c(c - \ell_{ij})(c - \ell_{jk})(c - \ell_{ki})} \tag{12}$$

Once we have the area A, we can divide by ℓ_{ij} to get the height h of the triangle, taking edge ij as the temporary base of this triangle. This is also the height that defines a right triangle with some base, this height, and edge ℓ_{ik} as the hypotenuse. Thus, we can find the length of the final edge in this new right triangle using the Pythagorean theorem: $a' = \sqrt{\ell_{ik}^2 - h^2}$. Figure 5 shows the placement of these variables.

The slope of ik with respect to ij can then be obtained using the coordinate pair (a', h) and transforming this into SCS coordinates. This is one of the slopes stored in edge ij. We repeat for all the other pairs of edges involved.

7.4.2 Tracing Query

At any point p, a tracing query computes the point q reached by walking a given distance $d > 0$ in a given direction u. More precisely, the input is a triangle ijk and the barycentric point p within the triangle, the edge ij (wlog) as a reference edge, the slope (m, s, r) denoting which direction to walk with respect to the reference edge, and d, the distance to travel. The output is another barycentric coordinate q within some triangle xyz.

First, during the algorithm we will have to compute the actual slope, since we will be solving line intersections. However, there are three different ways to describe a slope, depending on which edge of the triangle one uses as a reference, and we can change between them using slope difference from Section 7.2.1.

Therefore, the tracing query will first start by calculating the (m, s, r) coordinates of the inputted slope with respect to all three edges. It will then choose the best reference edge to work off of. Best reference edge is defined as the one where m is closest to the x-axis, if we let the reference edge be the x-axis. For this section only, m instead stands for actual slope, not the slope within the (m, s, r) coordinate system.

Without loss of generality, suppose the inputted reference edge is ij. We then set up the following coordinate system:

- The origin is vertex i.
- Vertex j is now located at $(\ell_{ij}, 0)$.
- Vertex k is now located using ℓ_{ik} and the slope of ik with respect to ij. This is done using the first operation described in Section 7.2.1.
- Convert barycentric coordinates p to this new coordinate system, which will have coordinates (x_p, y_p).

Therefore, the line that describes the trace is given by the equation $y - y_p = m(x - x_p)$. The line describing the ij edge in this system is trivially $y = 0$. The line describing the ik edge in this system can be found by taking the slope of ik with respect to ij and using vertex k as the point. The line describing the jk edge in this system can be found by taking the slope of jk with respect to ij and using vertex k as the point. What remains is to calculate the points of intersection of the lines defining the edges with the line defining the direction of travel, then picking the correct one based on the direction of travel. To do this, we can take the coordinate difference between the point of intersection and (x_p, y_p), which gets us a slope in Cartesian coordinates. We can then convert to (m, s, r) and see if it matches. Take the closest one that we find. See the pseudocode for more concrete steps.

Once we have the intersection point (and consequently, the triangle edge which has this intersection point), we then need to check the distance between this intersection point and the original point p. To do this, we convert this intersection point back to barycentric coordinates. We can then find the distance d' between p and and the intersection point using Equation 3.

If $d' = d$, then we can just return the triangle and intersection point that we found.

If $d' > d$, then we now have two points in the (x, y) plane, labelled (x_p, y_p) and (x_e, y_e), the distance between them d', and a distance d from one of the points towards the other. The final point in the (x, y) plane that we should return is then given as $\frac{d}{d'}(x_e - x_p, y_e - y_p)$. We can then convert this back to barycentric coordinates, then return this point and triangle ijk.

If $d' < d$, then we update d to become $d - d'$, then translate the slope so that the reference edge is the edge that we hit. We then move to the other triangle that uses this edge. In this other triangle, we now have the distance we want to travel, the point at which to start at (which is on an edge), the reference edge (the one that we hit), and the slope. We can then simply run this algorithm again.

Note that the line has to intersect at least two of the three edges at some point, while the third intersection might not exist, or be very numerically large. This third intersection is not a problem - if we detect that the lines are parallel, or that the result does not fit inside an integer, we can simply ignore this intersection point and only take the other two points into consideration. Therefore, this will not damage our numerical stability at all. For the other two points, a clear solution always exists.

There are a few edge cases to account for. If our direction of travel and point are such that we are travelling along an edge, then we will have an infinite number of intersection points. This is easy to detect, since the two lines will have the same slope and intercept. If this is the case, we can instead directly compare

the edge length with d, and either calculate the final point accordingly, or go to the vertex and enter a new triangle.

The other edge case is when our direction of travel causes us to hit a vertex directly. In this case, we will need to sweep around the vertex in order to figure out which triangle to go to next. This can be done by slope-chaining around the vertex one by one and updating the slope as we go along. An alternative way of viewing this is that our path will still pass through triangles, even if the the path only crosses a single vertex within that triangle.

7.4.3 Intrinsic Vertex Update

Sharp et al. [3] had a third atomic operation in order to update the vertices. However, our data structure does not handle vertices, instead choosing to bring edges to the forefront by using slopes. We no longer need this atomic operation.

7.5 Local Operations

7.5.1 Edge Flip

An edge flip replaces a pair of triangles ijk, jkl, both in F, with a new triangle pair ijl, ikl. The most difficult part here is to compute the length of the new edge il **without** the use of angles.

Take the new triangle ijl. We know edge lengths ij and jl from the original triangles. We also have the slope of jk with respect to ij and the slope of jl with respect to jk. Therefore, we can calculate the slope of jl with respect to ij using the slope chaining in Section 7.2.1.

We now have two side lengths of the triangle ijl (sides ij and jl), along with the angle between them (although this angle is represented by a slope). This is enough to fully define the triangle - so we can then find the length of the third side il. Let the slope of jl with respect to ij be denoted (m, s, r). We can convert this back into (x, y) form. This gives us x and y as the two sidelengths of a right triangle, with the hypotenuse being il. We can then use the Pythagorean theorem to get the edge length $il = \sqrt{x^2 + y^2}$. Once we have all the edge lengths, we can compute all the slopes for all the edges involved in the edge flip using the Triangle Update atomic operation (Section 7.4.1).

7.5.2 Face Split or Vertex Insertion

Given the barycentric coordinates for a point p in the interior of an intrinsic triangle ijk, a vertex insertion replaces triangle ijk with three new intrinsic triangles ijp, jkp, and kip. Therefore, the new edge lengths can be obtained using Equation 3. All the triangles can then be updated using the Triangle Update atomic operation (Section 7.4.1).

7.5.3 Other Operations

The other operations are edge splitting, intrinsic vertex removal and intrinsic vertex repositioning. However, these are all built using the operations already described above as black boxes in Sharp et al. [3] and Gillespie et al. [1]. Therefore, it is unnecessary to cover them here.

8 Conclusion

Therefore, in this report, we have outlined a new data structure, linearpost, that appears superior to signposts due to the avoidance of transcendental functions. This gives us more stable tracing queries, which in turn, can help us when performing vertex insertions in the integer-coordinate data structure as described in Section 6. This would have the logical best improvements to robustness, as we avoid all calls to transcendental functions. With linearpost hybridization, we have a data structure which makes no calls to transcendental functions, with only integers stored along each edges, would present the ideal data structure for working with intrinsic meshes in theory.

Gillespie et al. [1] determined that the technique of extracting intersections with extrinsic edges by tracing along an intrinsic edge can sometimes fail. We propose that, unless the mesh is adversarial, local

tracing queries will never fail in returning the correct *number* of crossings. This is in part due to the far more stable linearpost data structure, but also because even if the instability causes this tracing to return incorrect positions, the benefit of the normal coordinate data structure is that we do not care exactly where the intersections are located - we just want the number of them.

8.1 Future Work

The next step would be to implement each variation of the data structures and benchmark them, testing for numerical error and failures. The existing code for signposts is a simpler implementation, however the one used for benchmarking uses an alternative, fancier angle representation. We would seek to implement linearposts, hybridization with signposts, and hybridization with linearposts, and see how each perform with respect to numerics and speed.

Finally, there are a few other optimizations and potential avenues of exploration in the linearpost data structure. At the moment, each edge stores four slopes, one for each other edge that the original edge is connected to. It is still in question whether this can be reduced to one or two slopes. Finally, with the use of slopes and lengths, it appears that we've unlocked the realm of linear algebra. Perhaps there are linear algebra tools out there that can help speedup computation and efficiency, or perhaps open the door to run calculations in parallel.

References

[1] Mark Gillespie, Nicholas Sharp, and Keenan Crane. Integer coordinates for intrinsic geometry processing. *ACM Trans. Graph.*, 40(6), December 2021.

[2] Max Schindler and Evan Chen. Barycentric coordinates in olympiad geometry. May 2012.

[3] Nicholas Sharp, Yousuf Soliman, and Keenan Crane. Navigating intrinsic triangulations. *ACM Trans. Graph.*, 38(4), July 2019.

A Pseudocode

BaryToCart
Input: Triangle ijk, Cartesian coordinates of vertices i, j, and k, in the form (x_1, y_1), (x_2, y_2), and (x_3, y_3) and barycentric point $p = (p_1, p_2, p_3)$.
Output: Cartesian coordinates for point p.

BaryToCart:
$x_p = p_1x_1 + p_2x_2 + p_3x_3$
$y_p = p_1y_1 + p_2y_2 + p_3y_3$
return (x_p, y_p)

CartToBary
Input: Triangle ijk, Cartesian coordinates of vertices i, j, and k, in the form (x_1, y_1), (x_2, y_2), and (x_3, y_3) and Cartesian point $p = (x_p, y_p)$.
Output: Barycentric coordinates for point p.

CartToBary:
$p_1 = \frac{(y_2-y_3)(x_p-x_3)+(x_3-x_2)(y_p-y_3)}{(y_2-y_3)(x_1-x_3)+(x_3-x_2)(y_1-y_3)}$
$p_2 = \frac{(y_3-y_1)(x_p-x_3)+(x_1-x_3)(y_p-y_3)}{(y_2-y_3)(x_1-x_3)+(x_3-x_2)(y_1-y_3)}$
$p_3 = 1 - p_1 - p_2$
return (p_1, p_2, p_3)

BaryDistance
Input: Triangle ijk with associated edge lengths ℓ_{ij}, ℓ_{ik}, ℓ_{jk}, points p and q in barycentric coordinates.
Output: Euclidean distance between points p and q.

BaryDistance:
$u = p - q$
$d = \sqrt{-(\ell_{ij}^2u_1u_2 + \ell_{jk}^2u_2u_3 + \ell_{ki}^2u_3u_1)}$
return d

SlopeToRealSlope
Input: SCS coordinates (m, s, r).
Output: Real slope m' without s and r bits.

SlopeToRealSlope:
$sr = 2s + r$
switch sr:
  $sr = 0$: return m
  $sr = 1$: return $-\frac{1}{m}$
  $sr = 2$: return $-m$
  $sr = 3$: return $\frac{1}{m}$

SlopeToCart
Input: SCS coordinates (m, s, r).
Output: Corresponding Cartesian coordinates at unit distance from origin.

SlopeToCart:
$x' = \sqrt{\frac{1}{1+m^2}}$
$y' = \sqrt{\frac{m^2}{1+m^2}}$
$sr = 2s + r$
switch sr:
  $sr = 0$: return (x', y')
  $sr = 1$: return $(-y', x')$
  $sr = 2$: return $(-x', -y')$
  $sr = 3$: return $(y', -x')$

CartToSlope
Input: Cartesian coordinates (x, y).
Output: Corresponding SCS coordinates (m, s, r).

CartToSlope:
$s = (|x| \leq |y|)$
$r = (y \leq -x)$
if $r = 0$
  $m = \frac{y}{x}$
else
  $m = -\frac{x}{y}$
return (m, s, r)

SlopeToCartDist
Input: SCS coordinates (m, s, r), distance from origin ℓ
Output: Corresponding Cartesian coordinates.

SlopeToCartDist:
return $\ell \cdot$ ***SlopeToCart***$((m, s, r))$

SlopeChain
Input: Slope of il with respect to ik, denoted (m_{il}, s_{il}, r_{il}) and slope of ik with respect to ij, denoted (m_{ik}, s_{ik}, r_{ik}), all in SCS coordinates.
Output: Slope of il with respect to ij, in SCS coordinates.

SlopeChain:
$(x_{il}, y_{il}) =$ ***SlopeToCart***$((m_{il}, s_{il}, r_{il}))$
$(x_{ik}, y_{ik}) =$ ***SlopeToCart***$((m_{ik}, s_{ik}, r_{ik}))$
$\vec{x}_{ik} = (x_{ik}, y_{ik})$
extendedX $= (x_{ik}, y_{ik}, 0)$
extendedY $= (a, b, 0) =$
  CrossProduct(extendedX, $(0, 0, -1)$)
$\vec{y}_{ik} = (a, b)$
return $x_{il}\vec{x}_{ik} + y_{il}\vec{y}_{ik}$

SlopeDifference

Input: Slope of il with respect to ij, denoted (m_{il}, s_{il}, r_{il}) and slope of ik with respect to ij, denoted (m_{ik}, s_{ik}, r_{ik}), all in SCS coordinates.

Output: Slope of il with respect to ik, in SCS coordinates.

SlopeDifference:
$(x_{il}, y_{il}) =$ ***SlopeToCart***$((m_{il}, s_{il}, r_{il}))$
$(x_{ik}, y_{ik}) =$ ***SlopeToCart***$((m_{ik}, s_{ik}, r_{ik}))$
$\vec{x}_{ik} = (x_{ik}, y_{ik})$
extendedX $= (x_{ik}, y_{ik}, 0)$
extendedY $= (a, b, 0) =$
  CrossProduct(extendedX, $(0, 0, -1)$)
$\vec{y}_{ik} = (a, b)$
$\hat{\vec{x}}_{ik} = (x_{ik}, a)$
$\hat{\vec{y}}_{ik} = (y_{ik}, b)$
return $x_{il}\hat{\vec{x}}_{ik} + y_{il}\hat{\vec{y}}_{ik}$

TriangleUpdate

Input: Triangle ijk with side-lengths ℓ_{ij}, ℓ_{jk}, and ℓ_{ki}

Output: Edges ij, jk, and ki, where for each edge, the two slopes that deal with triangle ijk are recalculated.

TriangleUpdate:
Edges $= \{ij, jk, ki\}$
$c = \frac{\ell_{ij}+\ell_{jk}+\ell_{ki}}{2}$
$A = \sqrt{c(c-\ell_{ij})(c-\ell_{jk})(c-\ell ki)}$
for edge ab in Edges
  otherEdges = Edges without edge ab
  for edge $\alpha\beta$ in otherEdges
    $h = \frac{A}{\ell_{ab}}$
    $a' = \sqrt{\ell_{\alpha\beta}^2 - h^2}$
    $\alpha\beta$'s slope with respect to $ab =$
      CartToSlope$((a', h))$

TracingQuery

Input: Triangle ijk and associated data, barycentric point p in triangle ijk, reference edge $ab \in \{ij, jk, kl\}$, slope (m, s, r), distance to travel d

Output: Barycentric coordinate q in triangle xyz and triangle xyz.

TracingQuery:
while $d > 0$
  otherEdges $= \{ij, jk, kl\}$ without ab
  $(m_1, s_1, r_1) = (m, s, r)$
  for index i from 1 to 2
    $(m_{i+1}, s_{i+1}, r_{i+1}) =$
      SlopeDifference$((m, s, r)$,
        slope of otherEdges$[i+1]$ w.r.t. $ab)$
  $i =$ argmax of $|m_i|$ for i from 1 to 3
  if i is not 1
    $ab =$ otherEdges$[i-1]$
    $(m, s, r) = (m_i, s_i, r_i)$
  otherEdgeSlopes $= (m_j, s_j, r_j)$ for j from 1 to 3, excluding i
  $(x_a, y_a) =$ ***SlopeToCartDist***(otherEdgeSlopes[1])
  $(x_b, y_b) =$ ***SlopeToCartDist***(otherEdgeSlopes[2])
  $(x_p, y_p) =$ ***BaryToCart***$(p, (0,0), (x_a, y_a), (x_b, y_b))$

  $m' =$ ***SlopeToRealSlope***$((m, s, r))$
  $m'_1 =$ ***SlopeToRealSlope***(otherEdgeSlopes[1])
  $m'_2 =$ ***SlopeToRealSlope***(otherEdgeSlopes[2])
  $(x_\alpha, y_\alpha) =$ ***CalcLineIntersect***$(m', (x_p, y_p), 0, (0,0))$
  $(x_\beta, y_\beta) =$ ***CalcLineIntersect***$(m', (x_p, y_p), m'_1, (x_a, y_a))$
  $(x_\delta, y_\delta) =$ ***CalcLineIntersect***$(m', (x_p, y_p), m'_2, (x_b, y_b))$
  for index j in $\{\alpha, \beta, \delta\}$
    if ***CartToSlope***$((x_p - x_j, y_p - y_j)) = (m, s, r)$
      $(x_c, y_c) = (x_j, y_j)$
      $\gamma =$ edge that (x_c, y_c) is on
      break

  $d' =$ ***BaryDistance***(triangle ijk, p,
    CartToBary(triangle ijk, (x_c, y_c)))
  if $d' = d$
    return ***CartToBary***(triangle ijk, (x_c, y_c)), triangle ijk
  if $d' > d$
    $(x_f, y_f) = \frac{d}{d'}(x_c - x_p, y_c - y_p)$
    return ***CartToBary***(triangle ijk, (x_f, y_f)), triangle ijk
  if $d' < d$
    $d = d - d'$
    $(m, s, r) =$ ***SlopeDifference***$((m, s, r)$, slope of γ w.r.t. $ab)$
    Triangle $ijk =$ other triangle tnumericshat has γ as an edge.

EdgeFlip

Input: Triangles ijk and jkl and associated data.

Output: An updated linearpost data structure with new triangles ijl and ikl, and old triangles ijk, jkl deleted.

EdgeFlip:
Slope of jl w.r.t. $ij = (m, s, r) =$
  SlopeChain(slope of jl w.r.t. jk, slope of jk w.r.t. ij)
$(x, y) =$ ***SlopeToCartDist***$((m, s, r), \ell_{jl})$
$\ell_{il} = \sqrt{x^2 + y^2}$
TriangleUpdate(Triangle ijl)
TriangleUpdate(Triangle ikl)

InsertVertex
Input: Triangle ijk and associated data and barycentric coordinate p within ijk.
Output: An updated linearpost data structure with a new vertex at point p, associated edges inserted, and slopes calculated.

InsertVertex:
$\ell_{ip} =$ ***BaryDistance***(Triangle ijk, p, $(1, 0, 0)$)
$\ell_{jp} =$ ***BaryDistance***(Triangle ijk, p, $(0, 1, 0)$)
$\ell_{kp} =$ ***BaryDistance***(Triangle ijk, p, $(0, 0, 1)$)
Add triangle ijp, jkp, kip to the linearpost data structure.
TriangleUpdate(Triangle ijp)
TriangleUpdate(Triangle jkp)
TriangleUpdate(Triangle kip)