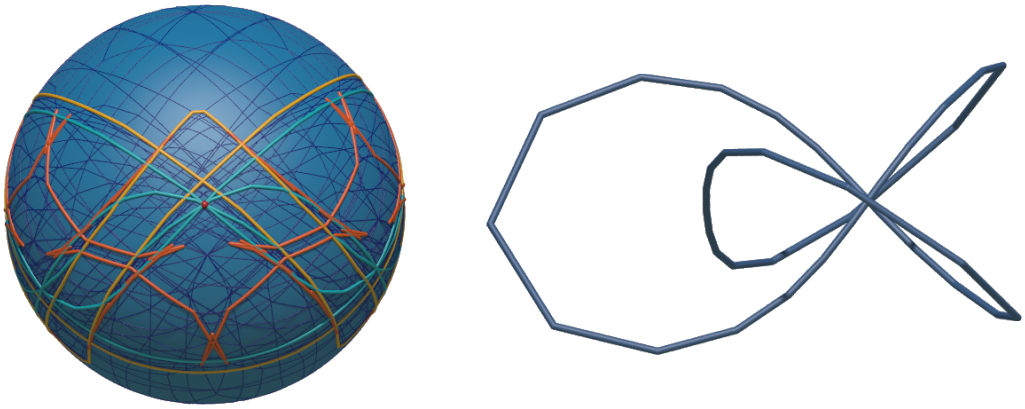
Technische Universität Berlin
Institut für Mathematik
Sullivan / Knöppel
http://www3.math.tu-berlin.de/geometrie/Lehre/SS17/MathVis SS 17

Mathematical Visualization

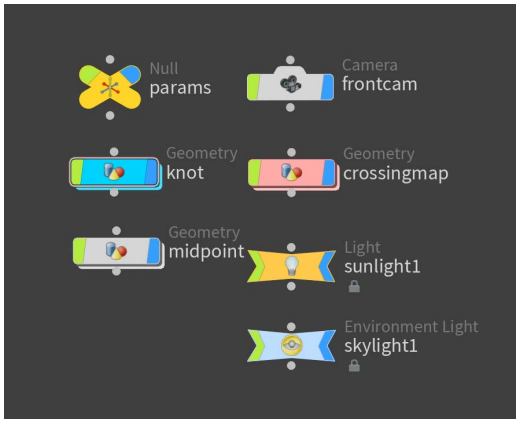
Assignment 5 - Crossing map[1]

Let γ be a knot. Let $U \subset \mathbb{S}^2$ denote the set of directions for which we obtain a regular knot diagram and V denote the set of direction for which the diagram is irregular. The partition U, V of $\mathbb{S}^2$ is called the *crossing map*. The set V can be split into three parts as shown in the picture below: The tantrix (yellow), the binix (orange) and the tertrix (turquoise).



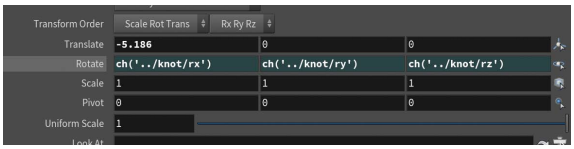
The thin blue curves are so called 0-curves - these directions correspond to regular diagrams in which a vertex is projected onto an edge but the crossing is transversal.

Network setup. We want to be able to rotate the knot γ while we see the corresponding crossing map. As the computational cost will be larger this time, we don't want to transform γ and compute everything from scratch. Therefore we put the knot and the crossing map into different *geometry object nodes* and synchronize the rotational part of their transformation matrices.

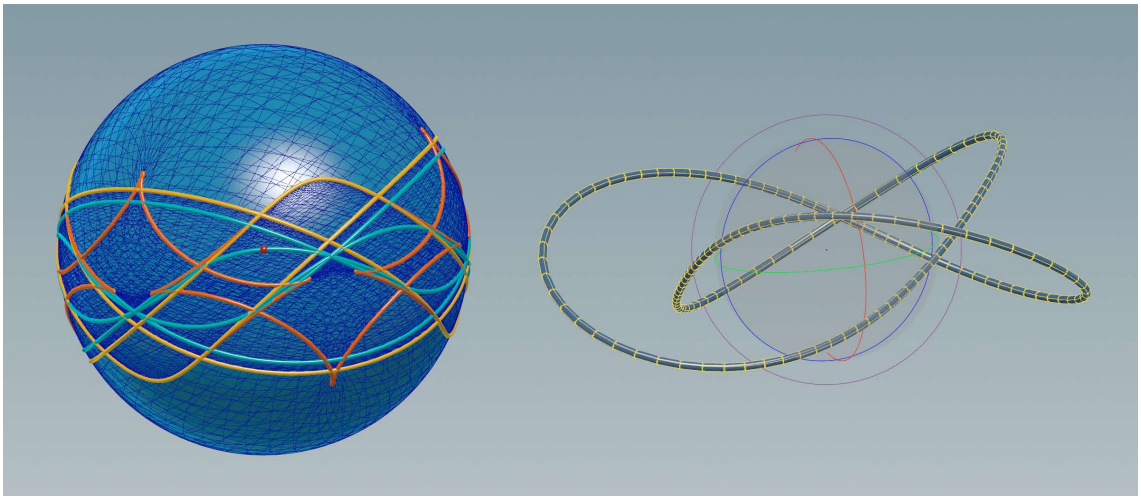


[1] due 18.6.2017

This can be done using the `ch` function to reference the rotation parameters `rx, ry, rz` of the *geometry object node* `knot` in the *geometry object node* `crossingmap`.

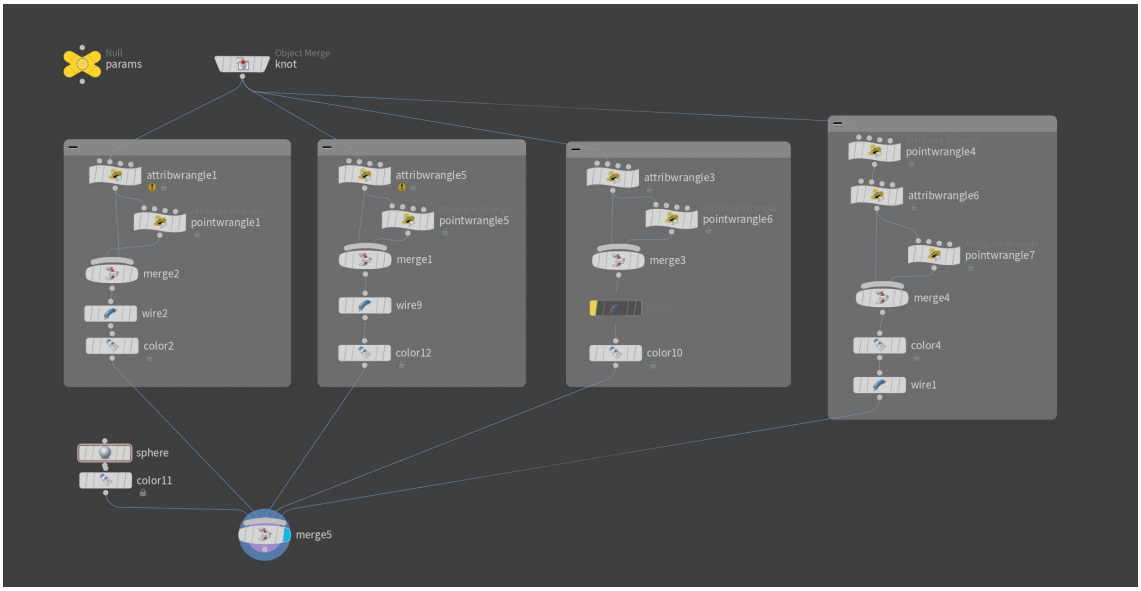


So we can use Houdini's *rotate* tool to rotate the knot and the crossing map is rotated accordingly.



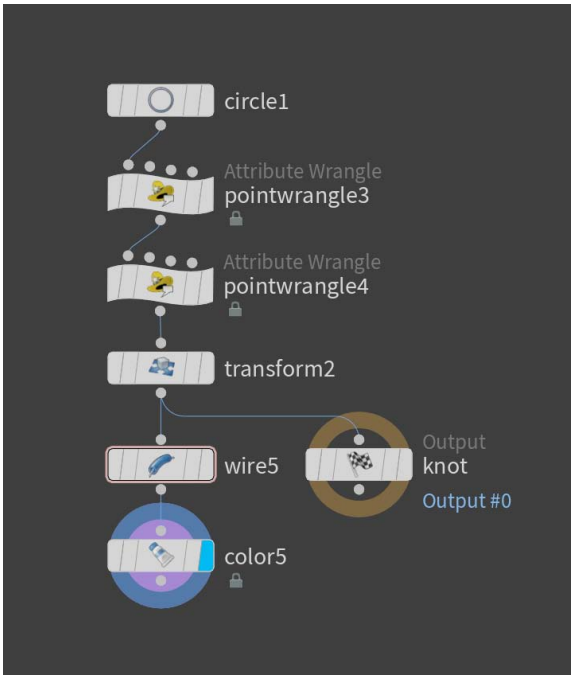
In particular we can use a fixed camera to look at the scene. Therefore we can easily mark the projection direction with a small red point on the sphere.

The crossing map is then computed by the following network contained in the *geometry object node* `crossing map`.

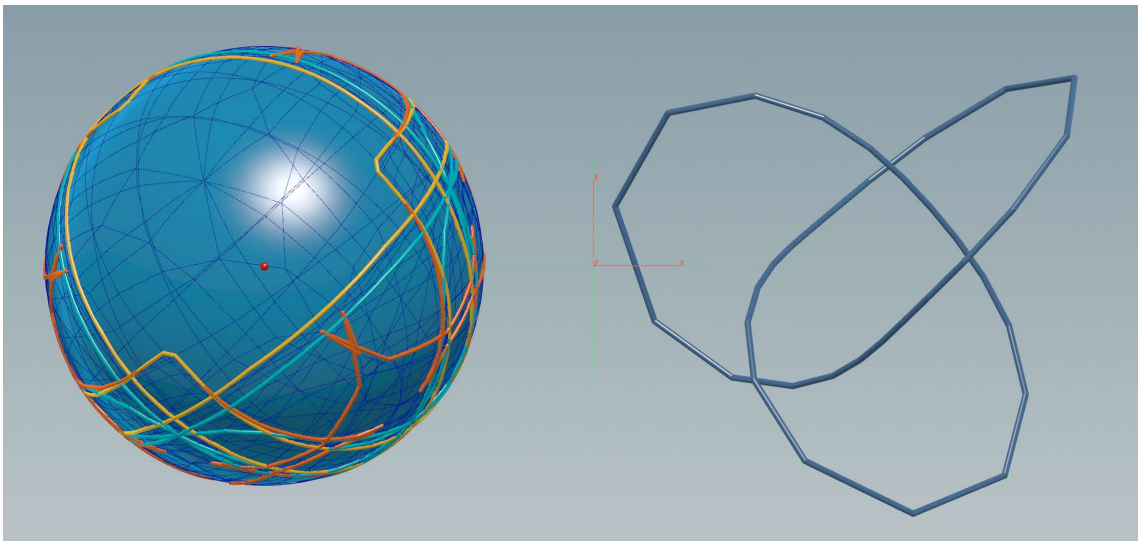


It basically consists of 4 parts - 3 to compute the parts of the crossing map as well as one to compute the 0-curves. Parameters are stored in a *null node*. The knot geometry is handed over

from the *geometry object node* `knot` by an *output channel node* (see the network below) and can be accessed by an *object merge node*.



0-curves. The 0-curves consist of the directions of secants of the knot γ, i.e a point on a 0-curve corresponds to a direction for which a vertex in the knot diagram lies on an edge.



These curves are easy to compute: We fix a point γ_i on the knot and look at all lines that intersect the edges of γ not incident to γ_i. This yields for each point γ_i two spherical polylines $\pm\eta^i$ on $\mathbb{S}^2$. The points of $\pm\eta^i_j$ are given by

$$\eta^i_j = \frac{\gamma_j - \gamma_i}{|\gamma_j - \gamma_i|}.$$

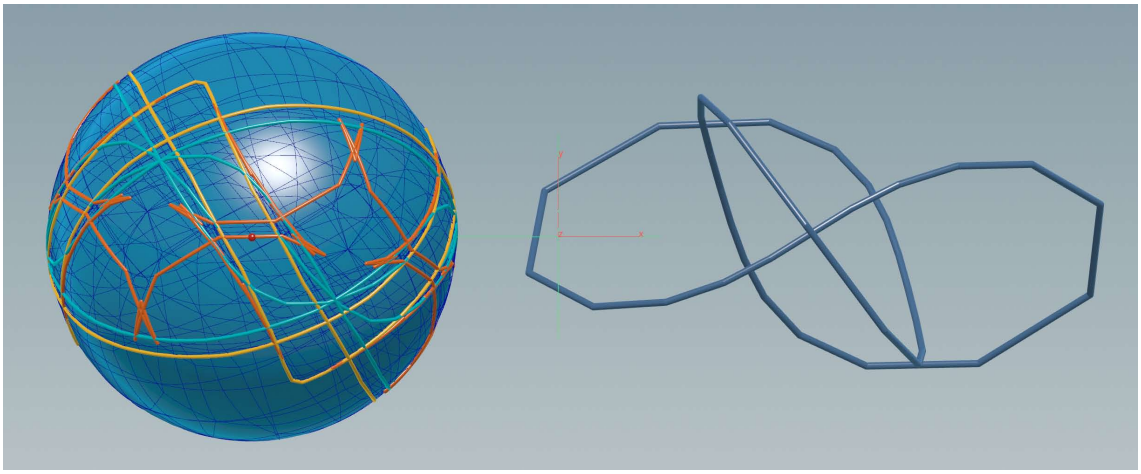
As for the tantrix we approximate $\pm\eta^i$ by Euclidean polylines. This can be achieved by subdividsion.

Algorithm 0-curves

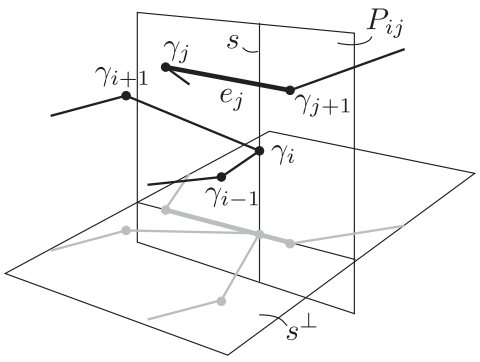
Input: polygon $\gamma = (\gamma_1, \ldots, \gamma_n)$, $d > 0$
Output: set of approximated 0-curves with all edge lengths $\leq d$

```
Z ← ∅;
for i = 1, ..., n do
    η ← empty polyline;
    for j = i + 1, ..., j + n − 1 do
        add (γ_{j%n} − γ_i) / |γ_{j%n} − γ_i| to η;
    end for
    while maximal edge length of η ≥ d do
        subdivide η;                          ▷ subdivide on the sphere S^2
    end while
    add ±η to Z;
end for
```

Since the edges of a spherical polyline consist of great circle arcs, we can directly subdivide when adding the points - just as we did in the last assignment to subdivide the tantrix.



Binix. The binix consists of those parts the zero curves for which the corresponding knot diagram has a non-transversal intersection: If we consider a secant s through γ_i and the edge $e_j = (\gamma_j, \gamma_{j+1})$, then the direction of s belongs to the binix if γ_{i-1} and γ_{i+1} lie on the same side of the affine plane P_{ij} through γ_i and e_j.



For each γ_i we then successively collect those directions going along the edges of $\gamma \setminus \{e_{i-1}, e_i\}$:

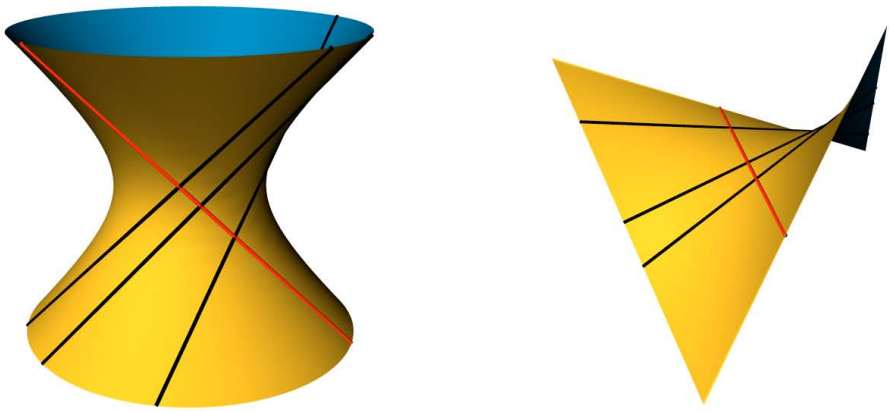
Algorithm Binix

Input: polygon $\gamma = (\gamma_1, \ldots, \gamma_n)$, $d > 0$
Output: binix with all edge lengths $\leq d$

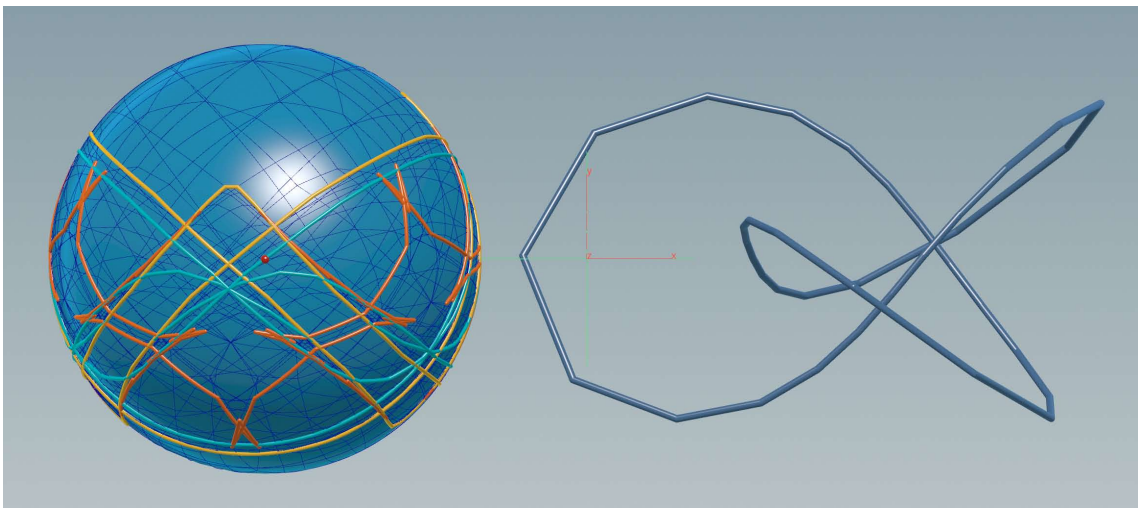
```
B ← ∅;
for i = 1, ..., n do
    j ← i + 1
    while j < i + n − 1 do;
        if γ_{(i−1)%n}, γ_{(i+1)%n} on same side of P_ij then
            η ← empty polyline;
            add (γ_{j%n} − γ_i)/|γ_{j%n} − γ_i| to η;          ▷ add start point of j-th edge
            while γ_{(i−1)%n}, γ_{(i+1)%n} on same side of P_ij do   ▷ while non-transversal
                add (γ_{(j+1)%n} − γ_i)/|γ_{(j+1)%n} − γ_i| to η;   ▷ add end point of j-th edge
                if j < i + n − 1 then
                    j ← j + 1;
                else
                    break;                                    ▷ stop if j-th edge is incident to γ_i
                end if
            end while
            while maximal edge length of η ≥ d do
                subdivide η;                                  ▷ subdivide on the sphere S^2
            end while
            add ±η to B;
        else
            j ← j + 1;
        end if
    end while
end for
```

Tertrix. The tertrix consists of trisecant directions, i.e. directions for which we see a triple crossing in the diagram.

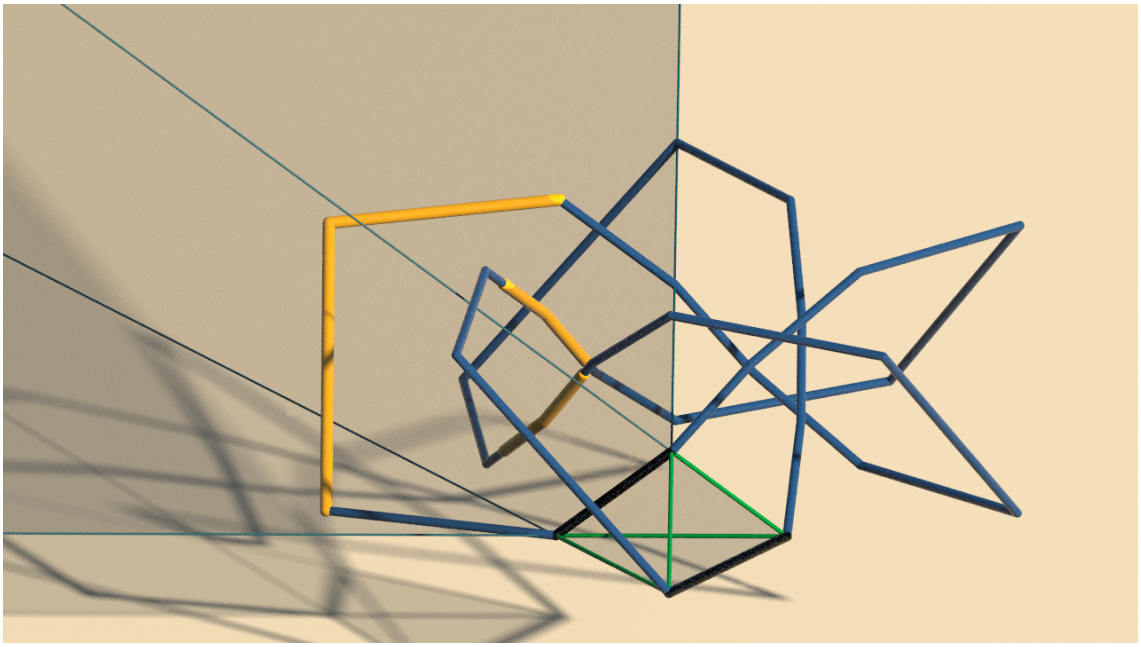
Given three pairwise skew edges e_i, e_j, e_k, then the set of lines intersecting all three edges is either empty or forms an interval of lines on a doubly ruled surface (below a picture of the situation took from Martin Sommer's thesis on *Visualization in Geometric Knot Theory*).



The corresponding segment of the tertrix are algebraic curves on $\mathbb{S}^2$. We can again use subdivision to approximate these segments.



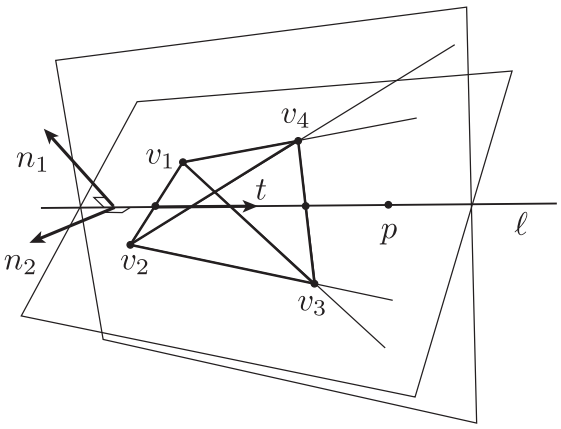
To compute the trisecant directions it is convenient to fix a pair of (skew) non-adjacent edges e_i and e_j of the knot γ.



The planes corresponding to the faces of the tetrahedron $(\gamma_i, \gamma_{i+1}, \gamma_j, \gamma_{j+1})$ split space into several regions. Three of these are interesting for us: the wedge containing e_i, the tetrahedron and the wedge containing e_j. For each point p in these 3 regions we always find a line through p which intersects both of the edges e_i and e_j. Thus for all points of the knot γ contained these regions we find a trisecant. Running over all non-adjacent pairs of edges we then find all trisecants of γ. Actually it is enough to consider just one of the three regions (see the picture above): Let W_{ij} denote the wedge of the tetrahedron $(\gamma_i, \gamma_{i+1}, \gamma_j, \gamma_{j+1})$ containing the edge $e_j = (\gamma_j, \gamma_{j+1})$.

The wedge is the intersection of 4 half spaces. The intersection of an edge and the wedge can be computed then by intersecting the edge with these half spaces successively.

Given $p \in W_{ij}$ there is a unique line through p which intersects both of the edges e_i and e_j. There are two corresponding directions one of which points 'from e_i to e_j'. Call this direction $t_{ij}(p)$. To compute it, consider the following picture of a point p which lies in the wedge of of the tetrahedron (v_1, v_2, v_3, v_4) that contains the edge (v_3, v_4):



Since the line ℓ lies in both planes, the plane through p, v_1, v_2 and the plane through p, v_3, v_4, the direction of ℓ is perpendicular to both their normals. The direction pointing from (v_1, v_2) to (v_3, v_4) is then given by

$$t = \frac{n_1 \times n_2}{|n_1 \times n_2|}, \quad \textit{where} \quad n_1 = \frac{(v_1 - p) \times (v_2 - p)}{|(v_1 - p) \times (v_2 - p)|}, \quad n_2 = \frac{(v_4 - p) \times (v_3 - p)}{|(v_4 - p) \times (v_3 - p)|}.$$

Note that the orientation of the tetrahedron play role here. If v_3 and v_4 are interchanged we have to change the sign of t.

Below you find the algorithm to compute the tertrix. This time the subdivision part looks different. This is due to the fact that the curve segments of the tertrix are no great circle arcs but (up to sign) given by the algebraic formula above. Thus we can just linearly subdivide the edges of the curve segments in the wedge until the image of its edge end points under t_{ij} are close enough.

Algorithm Tertrix

Input: polygon $\gamma = (\gamma_1, \ldots, \gamma_n)$, $d > 0$
Output: tertrix with all edge lengths $\leq d$

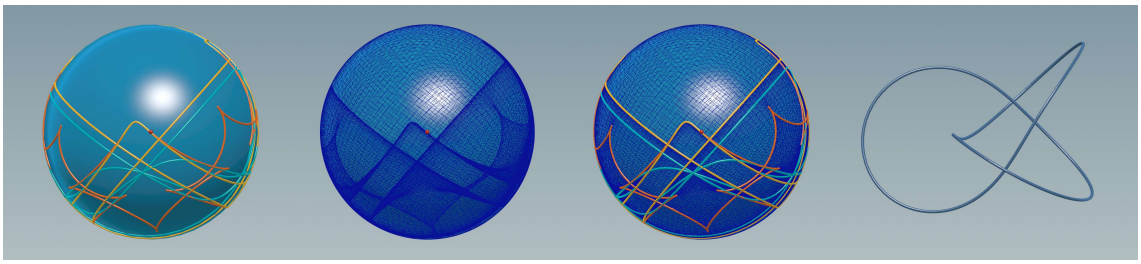
```
T ← empty set of curves;
for all edges e_i = γ_{i+1} − γ_i do
    for all edges e_j = γ_{j+1} − γ_i skew, non-adjacent to e_i do
        S ← list of curve segments of γ \ {e_{j−1}, e_j, e_{j+1}} intersecting W_ij;
        for all curves γ̃ in S do
            η ← t_ij(γ̃);                          ▷ apply t_ij to all points of γ̃
            while maximal edge length of η ≥ d do
                subdivide γ̃;
                η ← t_ij(γ̃);
            end while
            add η to T;
        end for
    end for
end for
```

The VEX function `addVertex` adds a vertex to the end of the polyline - it cannot insert a vertex at a different position. Thus we first store the points of $\tilde{\gamma}$ and $t(\tilde{\gamma})$ in a vector-array, where we can insert elements at arbitrary positions. Then we can do the subdivision on the level of arrays. Finally we create the polyline and copy the points to it. Below an implementation in VEX:

```
float maxdist = chf('maxdist');
int n = npoints(1);

for(int i=0; i<n; i++){
    for(int j=i+2; j<i+n-3; j++){

        // vertex points of the tetrahedron
        vector v1 = attrib(1,'point','P',i);
        vector v2 = attrib(1,'point','P',(i+1)%n);
        vector v3 = attrib(1,'point','P',j%n);
        vector v4 = attrib(1,'point','P',(j+1)%n);

        float det = dot(v2-v1,cross(v3-v1,v4-v1));

        if(abs(det)>1E-6){// if the edges are skew do:

            if(det<0){ // we assume the tetrahedron is positively
               oriented
                vector tmp = v1;
                v1=v2;
                v2=tmp;
            }

            // compute tertrix segments in current wedge
            int start = j+2;
            int k = start;
            while(k<start+n-3){
                // start and end point of current edge
                vector p = attrib(1,'point','P',k%n);
                vector q = attrib(1,'point','P',(k+1)%n);
                intersectWedge(v1,v2,v3,v4,p,q);
                // if edge intersects wedge continue gamma
                // until gamma leaves the wedge again and
                // get the corresponding tertrix directions
                vector gammaseg[] = {};
                vector tertrixseg[] = {};
                float dist[] = {}; // needed for subdivision later
                if(distance(p,q)>0){
                    vector tp = t(v1,v2,v3,v4,p);
                    push(gammaseg,p);
                    push(tertrixseg,tp);
                    while(distance(p,q)>0){
                        vector tq = t(v1,v2,v3,v4,q);
                        float d = acos(dot(tp,tq));
                        push(gammaseg,q);
                        push(tertrixseg,tq);
                        push(dist,d);
                        k++;
                        p = attrib(1,'point','P',k%n);
                        q = attrib(1,'point','P',(k+1)%n);
                        tp = tq;
                        intersectWedge(v1,v2,v3,v4,p,q);
                        if(k>=start+n-3) break;
                    }
                }else{
                    k++;
                }
```

```
                // subdivide until each edge length is smaller than
                // the given maxlength
                int maxIter = 50; // stop after 50 iteration
                int numIter = 0; // counter
                while(max(dist)>maxdist){
                    // subdivide
                    foreach(int k; float d; dist)
                        if(d>maxdist){
                            vector gamma_ = (gammaseg[k]+gammaseg[k+1])
                                /2.;
                            insert(gammaseg,k+1,gamma_);
                            vector t_k = tertrixseg[k];
                            vector t_kp = tertrixseg[k+1];
                            vector t_ = t(v1,v2,v3,v4,gamma_);
                            insert(tertrixseg,k+1,t_);
                            float d1 = acos(dot(t_k,t_));
                            float d2 = acos(dot(t_,t_kp));
                            dist[k] = d1;
                            insert(dist,k+1,d2);
                        }
                    // stop if a maximal number of iterations
                    // is reached
                    numIter++;
                    if(maxIter<numIter) break;
                }

                // add the corresponding polyline to the geometry
                int segment = addprim(0,'polyline');
                foreach(vector t; tertrixseg)
                    addvertex(0,segment,addpoint(0,t));

            }
        }
    }
}
```

Here the method `intersectWedge(vector a,b,c,d; export vector p,q)` intersects the edge (p, q) with the wedge of the simplex (a, b, c, d) opposite to the edge (a, b) and stores the endpoints of the intersection in `p` and `q` (if the intersection is empty it returns a degenerate edge, `p=q`). The method `t(vector a,b,c,d; vector p)` returns the direction of the line through p intersecting the edges (a, b) and (c, d) [pointing from (a, b) to (c, d)].



Exercise 5: Wire up a network to visualize the crossing map. As described above, compute the tantrix, binix and tertrix for a given knot. Prepare a setup that shows the knot diagram and the crossing map next to each other.