VISUALIZING KNOTS

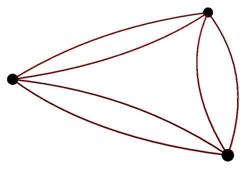
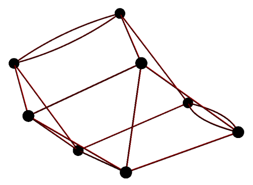
With this project I try to plot any knot from only the data of its crossings. The idea is to first compute the position of the crossings by computing the vertex positions of a graph corresponding to the knot. Then a vertex has to be deformed into a crossing and one has to connect the right parts of the crossings with each other. I will give a more detailed description of each step below. The starting point are the following knots *t[1]...t[5]* which I found in the lecture material. *t[1]* is a trefoil and *t[5]* is the 48-crossing knot that we couldn't compute the Jones Polynomial of with our 'naive' function from the very first lecture.
I will explain all nontrivial functions in this project either with comments in the code or in the text above or below the functions.

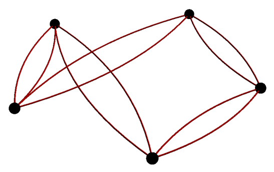
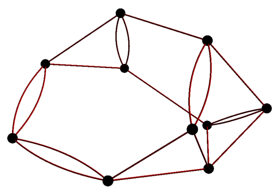
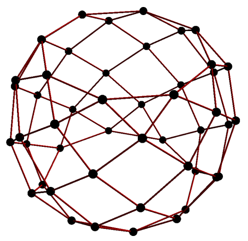
```
t[1] = Xp[1, 5, 2, 4] Xp[5, 3, 6, 2] Xp[3, 1, 4, 6];
t[2] = Xm[1, 12, 2, 13] Xm[7, 2, 8, 3] Xm[3, 8, 4, 9] Xm[11, 4, 12, 5]
    Xp[13, 7, 14, 6] Xp[9, 15, 10, 14] Xp[15, 11, 16, 10] Xp[5, 1, 6, 16];
t[3] = Xm[1, 6, 2, 7] Xm[3, 8, 4, 9] Xm[5, 10, 6, 1] Xm[7, 2, 8, 3] Xm[9, 4, 10, 5];
t[4] = Xm[5, 12, 6, 13] Xm[9, 16, 10, 17] Xm[11, 6, 12, 7]
    Xm[13, 20, 14, 1] Xm[15, 18, 16, 19] Xm[17, 10, 18, 11]
    Xm[19, 14, 20, 15] Xp[2, 8, 3, 7] Xp[4, 2, 5, 1] Xp[8, 4, 9, 3];
t[5] = Xp[4, 88, 5, 87] Xp[5, 75, 6, 74] Xp[6, 62, 7, 61] Xp[7, 49, 8, 48]
    Xp[8, 36, 9, 35] Xp[9, 23, 10, 22] Xp[16, 4, 17, 3] Xp[17, 87, 18, 86]
    Xp[18, 74, 19, 73] Xp[19, 61, 20, 60] Xp[20, 48, 21, 47] Xp[21, 35, 22, 34]
    Xp[28, 16, 29, 15] Xp[29, 3, 30, 2] Xp[30, 86, 31, 85] Xp[31, 73, 32, 72]
    Xp[32, 60, 33, 59] Xp[33, 47, 34, 46] Xp[40, 28, 41, 27] Xp[41, 15, 42, 14]
    Xp[42, 2, 43, 1] Xp[43, 85, 44, 84] Xp[44, 72, 45, 71] Xp[45, 59, 46, 58]
    Xp[52, 40, 53, 39] Xp[53, 27, 54, 26] Xp[54, 14, 55, 13] Xp[55, 1, 56, 96]
    Xp[56, 84, 57, 83] Xp[57, 71, 58, 70] Xp[64, 52, 65, 51] Xp[65, 39, 66, 38]
    Xp[66, 26, 67, 25] Xp[67, 13, 68, 12] Xp[68, 96, 69, 95] Xp[69, 83, 70, 82]
    Xp[76, 64, 77, 63] Xp[77, 51, 78, 50] Xp[78, 38, 79, 37] Xp[79, 25, 80, 24]
    Xp[80, 12, 81, 11] Xp[81, 95, 82, 94] Xp[88, 76, 89, 75] Xp[89, 63, 90, 62]
    Xp[90, 50, 91, 49] Xp[91, 37, 92, 36] Xp[92, 24, 93, 23] Xp[93, 11, 94, 10];
```

Associated Graph

Given any knot data it is possible to find an associated graph in which each crossing is vertex and vertices which correpond to crossings sharing an arc in the knot are connected with an edge. Our convention in class was to label each arc of a knot with natural numbers starting with 1. Each Crossing was labeled with either *Xp[i,j,k,l]* or *Xm[i,j,k,l]*, depending on its orientation with the four labels of the arcs coming together at this crossing starting with the incoming lower arc and proceeding counterclockwise. Hence the first label of the incoming lower arc is a unique label of each crossing and later for its corresponding vertex. The following function takes advantage of this fact and uses the *Intersection* function to find the number of equal labels of two crossings to find the number of arcs between them.

```
VisualizeKnot[knot_] := Module[{a, b, c, d, e},
                                         a = knot /. Times → List;
                                         b = Subsets[a, {2}] /. {Xm → List , Xp → List};

   c = List[#[[1, 1]], #[[2, 1]], Length[#[[1]] ∩ #[[2]]]] & /@ b;
                                         d = DeleteCases[c, {x_, y_, 0}] ;

   e = d /. {x_, y_, 2} ⧴ Sequence[{x, y, 1}, {x, y, 1}] /. {x_, y_, 1} ⧴ x ⟷ y;

   Graph3D[e, VertexStyle → Black, EdgeStyle → Red]
                                 ];

Table[VisualizeKnot[t[i]], {i, 5}]
```

{  ,  ,

 ,  ,  }

To avoid confusion between different labels of the vertices, I will refer in the following to the label of a vertex that comes from the association of the function above as *natural labels*. The natural labels of the trefoil are for example {1 ⟷ 3, 1 ⟷ 3, 1 ⟷ 5, 1 ⟷ 5, 3 ⟷ 5, 3 ⟷ 5}. However for later purposes it is convenient to introduce *canonical labels* in which the vertices are relabeld with numbers 1 ... *n* in order of their first appearence in the list of vertices. Later I will also introduce *split labels.*

The function *FindConnection* is just a copy of some code from above to directly use its output as an input for the function *CreateAdjacency*. It outputs a list of lists which are of the form {vertex_label *A*, vertex_label *B*, number of connections} . As the name suggests, the function *CreateAdja* *cency* creates the adjacency matrix of the graph corresponding to the knot, but without using the graph function, but the output of *FindConnections*. Now it is important to use *canonical labels* because both this adjacency matrix and the order in which mathematica considers the labels of a graph outputted by *VisualizeKnot* are exactly the the *canonical labels*, meaning the label of vertices is the order of appearance. To easily transform between them *CreateAdjacency* also keeps track of this relabeling and creates an association for this transformation.

```
FindConnections[knot_] := Module[{a, b, c, d},
                                  a = knot /. {Times → List, Xm → List , Xp → List};
                                  b = Subsets[a, {2}];

   c = List[#[[1, 1]], #[[2, 1]], Length[#[[1]] ⋂ #[[2]]]] & /@ b;
                                  d = DeleteCases[c, {x_, y_, 0}]
                                ];
```

```
x = FindConnections[t[1]]
```

```
{{1, 3, 2}, {1, 5, 2}, {3, 5, 2}}
```

```
CreateAdjacency[x_] := Module[{m = {{0}}, j = 1, a = Association[{}]},

   AddMore[n_] := Append[Append[#, 0] & /@ n, Table[0, {i, Dimensions[n][[2]] + 1}]];
                                      Do[Do[If[! ContainsAll[Keys[a], {i[[k]]}],
      AssociateTo[a, i[[k]] → j] If[j ≠ 1, m = AddMore[m]] j++;], {k, 2}];

    m[[ a[i[[1]]], a[i[[2]]]]] = i[[3]];, {i, x}];
                                      m = m + Transpose[m];
                                      {m, a}
                                ];
```

```
CreateAdjacency[x][[1]] // MatrixForm
```

$$\begin{pmatrix} 0 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 2 & 0 \end{pmatrix}$$

This is a very good example to explain *canonical labels*. In the expression {1 ⟷ 3, 1 ⟷ 3, 1 ⟷ 5, 1 ⟷ 5, 3 ⟷ 5, 3 ⟷ 5} the 1 appears first, so it's labeled with 1. Then 3 appears second, so it's labeled with 2. Then 5 appears to it's labeled with 3.

```
CreateAdjacency[x][[2]]
```

```
<|1 → 1, 3 → 2, 5 → 3|>
```

Splitting The Vertices

Since the numbers of each crossing are picked up by a counterclockwise path around the crossing, it is clear that the first one corresponds always to an incoming lower arc, the second one always an upper

arc which is incoing in the case of a left handed crossing and outgoing in the case of a right handed crossing, and so forth.
By examining the label, which causes the intersection that was used in the function *VisualizeKnot* above to find that there exists an arch between two vertices in the first place, the following function computes if this particular arc is part of the lower or upper part of the crossing. The output is a list of elements of the form {vertex_label *B*, +−1, vertex_label *A*, +−1} indicating that there exists an arc between the *upper (= +1)* or *lower (= -1)* part of vertex *A* and the *upper* or *lower* part of vertex *B* in terms of natural labels. This plus the handedness of each crossing is already enough information to recreate a knot.

```
SplitKnot[knot_] := Module[{a, b, c, d, d2, d3, d4, d5},
                                 a = knot /. {Times → List, Xm → List , Xp → List};
                                 b = Subsets[a, {2}];

   c = List[#[[1, 1]], #[[2, 1]], #[[1]] ⋂ #[[2]]] & /@ b;
                                 d = DeleteCases[c, {x_, y_, {}}];

   d2 = Table[Append[Take[#, 2], i], {i, Last[#]}] & /@ d;
                                 d3 = Flatten[d2, 1];
                                 d4 =
    (Take[#, 2] ~ Join ~ Mod[Position[Flatten[a], Last[#]], 4, 1] // Flatten) & /@ d3;
                                 d5 = d4 /. {i_, j_, k_, l_} :→ {i, (-1)^k, j, (-1)^l}
                            ];
```

```
SplitKnot[t[2]]
```

```
{{1, -1, 7, 1}, {1, 1, 11, -1}, {1, -1, 5, 1}, {1, 1, 13, -1}, {3, -1, 7, 1}, {3, 1, 7, -1},
 {3, -1, 11, 1}, {3, 1, 9, -1}, {7, -1, 13, 1}, {11, 1, 5, -1}, {11, -1, 15, 1},
 {5, -1, 13, 1}, {5, 1, 15, -1}, {9, 1, 13, -1}, {9, -1, 15, 1}, {9, 1, 15, -1}}
```

The function *CreateAdjacency* wasn't really helpful, because it was doing something that we already had, just in a different way. But with only slight modifications one can write a function *CreateSplitAdjacency* that does almost the same thing, except it treats the lower and upper part of each crossing in the knot as two different vertices in the graph. In addition it also keeps again track of the relabeling and outputs an association to transform from natural labels to canonical labels. For the sake of simplicity I'm using the old labels of each crossing for its upper part and 'minus' this label for the lower part.

```
CreateSplitAdjacency[knot_] := Module[{b, m = {{0}}, j = 1, a = Association[{}]},

   AddMore[n_] := Append[Append[#, 0] & /@ n, Table[0, {i, Dimensions[n][[2]] + 1}]];
                              b = SplitKnot[knot];

   Do[Do[If[! ContainsAll[Keys[a], {i[[k]] * i[[k + 1]]}], AssociateTo[a,
          i[[k]] * i[[k + 1]] → j] If[j ≠ 1, m = AddMore[m]] j++;], {k, {1, 3}}];
                                    m[[ a[i[[1]] * i[[2]]], a[i[[3]] * i[[4]]]]] = 1;,
    {i, b}];
                              m = m + Transpose[m];
                              {m, a}
                                    ];
```

```
CreateSplitAdjacency[t[1]][[1]] // MatrixForm
```

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

This example of the trefoil shows that for example the entry 1 in the split adjacency matrix in row 3 and column 4 corresponds to an edge between the vertices corresponding to the upper part of crossing 1 (1 → 3) and the lower part of crossing 3 (-3 → 4).

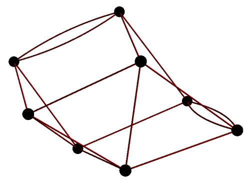
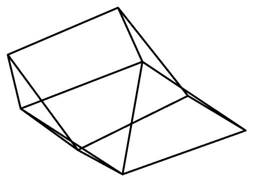
```
CreateSplitAdjacency[t[1]][[2]]
```

```
<|-1 → 1, 3 → 2, 1 → 3, -3 → 4, 5 → 5, -5 → 6|>
```

This is another function that seems to be useless, because all it does it taking a graph and producing the very same graph. The only difference is that the output is a graphics object consisting of lines and its edge lists does not contain duplicates. Later you will see that this notion of creating a graph with the *Graphics3D* function will be very useful.

```
ReproduceGraph[g_Graph] := Module[{v, a, d, b, c},

   v = Round[ VertexCoordinates /. AbsoluteOptions[g, VertexCoordinates], 0.1];
                                    a = AdjacencyMatrix[g];
                                    d = Dimensions[a][[1]];
                                    b = Flatten[
     Table[If[a[[i, j]] ≥ 1, Line[List[v[[i]], v[[j]]]], {}], {i, d}, {j, d}], 1];
                                    c = DeleteCases[b, {}];
                              Graphics3D[c, Boxed → False]
                              ];
```

```
List[VisualizeKnot[t[2]], ReproduceGraph[VisualizeKnot[t[2]]]]
```

{  ,  }

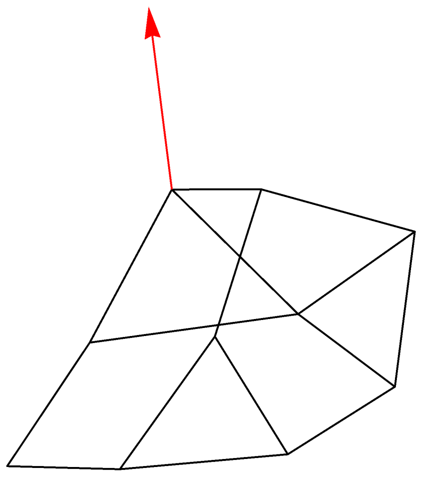
Localizing The Splitting

Now we come to the probably most difficult part of this project. As described above, each crossing has two parts to it, the upper and lower part. The function *SplitKnot* tells us moreover how edges connect those upper and lower parts, but in order to draw this as a graphics object, we'd also need to know how to separate the vertex, whichs vertex coordinate we already know, into two different vertices. If the graph is entirely embedded into a plane, the vertices can be split in the direction orthogonal to this plance. This is for example the case for the trefoil. In the general case, this direction has to be computed locally for each vertex. This is exactly what the following function *LocalSplittingDirections* is doing. Its output is a list of vectors corresponding to the direction in which the vertex should be split to maximize the distance between all edges. This is necessary because splitting a vertex along the direction of an edge would not show a clear distinction between upper and lower part of the crossing. The following function only computes the splitting direction up to +-1. It uses, depending on the number of neighbors of each vertex, either just the cross product between the vectors along the edges (for example in the case of the trefoil), or the *directed mean* of cross products of all possible combinations of vectors along edges. The *directed mean* evaluates a family of vectors in such a way that vectors never cancel. It is always adding up the each vector in the direction such that the signed overlap with the first vector in the family is maximal. The only downside is that the mean direction is only right up to sign.

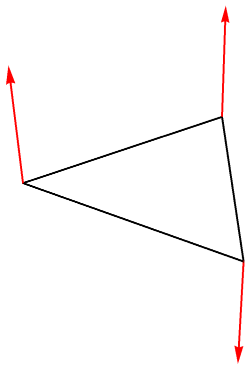
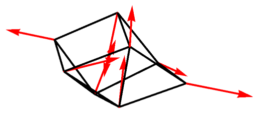
```
LocalSplittingDirections[knot_] := Module[{a, v, conn, relabel = Association[{}],
    j = 1, pairs, neighbors1, neighbors2, neighbors, directions},
                                                  a = VisualizeKnot[knot];

   v = Round[ VertexCoordinates /. AbsoluteOptions[a, VertexCoordinates], 0.1];
                                                  conn = FindConnections[knot];

   Do[Do[If[! ContainsAll[Keys[relabel], {i[[k]]}],
      AssociateTo[relabel, i[[k]] → j]; j++;], {k, {1, 2}}], {i, conn}];
                                                  pairs = Take[#, 2] & /@ conn;
                                                  neighbors1 =
    Table[{i}~Join~List[DeleteCases[Table[If[MemberQ[p, i], If[First[p] == i,
           Last[p], First[p]]], {p, pairs}], Null]], {i, Keys[relabel]}];
neighbors2 = neighbors1 /. relabel;

neighbors = Last[#] & /@ neighbors2;


DirectedMean[s_] := 1/Length[s] Sum[Sign[s[[1]].s[[i]]] * s[[i]], {i, Length[s]}];
                                                directions = Table[

  If[Length[neighbors[[q]]] == 2,
                                                            Mean[
    Flatten[Table[Cross[v[[i]] - v[[q]], v[[j]] - v[[q]]], {i, neighbors[[q]]},

      {j, Drop[neighbors[[q]], Det[Position[neighbors[[q]], i]]]}], 1]],

   If[Length[neighbors[[q]]] == 3,

    Mean[Flatten[Table[Normalize[Cross[v[[i1]] - v[[i2]], v[[i1]] - v[[i3]]]],
       {i1, neighbors[[q]]},

       {i2, Drop[neighbors[[q]], Det[Position[neighbors[[q]], i1]]]},

       {i3, Drop[neighbors[[q]], Det[Position[neighbors[[q]], i2]]]}], 2]],

    DirectedMean[Flatten[Table[Normalize[
        Cross[v[[i1]] - v[[i2]], v[[i1]] - v[[i3]]]], {i1, neighbors[[q]]},

       {i2, Drop[neighbors[[q]], Det[Position[neighbors[[q]], i1]]]},

       {i3, Drop[neighbors[[q]], Det[Position[neighbors[[q]], i2]]]}], 2]]
                                                       ]]
                                                        ,
  {q, Length[neighbors]}];
                                                Normalize /@ directions
                                         ];
```

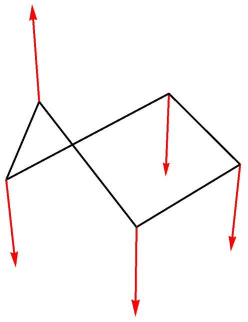
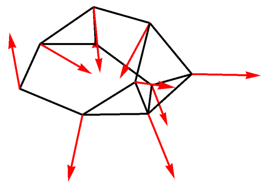
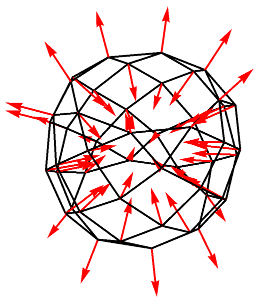
These two function draw this local splitting directions indicated at each vertex or only the nth vertex. You can see very well in the following graphics that the direction of this arrow minimizes its overlap with each of the edges attached to the vertex.

```
DrawLocalSplit[knot_] := Module[{pos, shift},
                                    pos = Round[ VertexCoordinates /.
      AbsoluteOptions[VisualizeKnot[knot], VertexCoordinates], 0.1];
                                    shift = LocalSplittingDirections[knot];

   Show[ReproduceGraph[VisualizeKnot[knot]], Graphics3D[
     {Red, Table[Arrow[{pos[[i]], pos[[i]] + shift[[i]]}], {i, Length[pos]}]}]]
                            ];

DrawLocalSplit[knot_, n_] := Module[{pos, shift},
                                    pos = Round[ VertexCoordinates /.
      AbsoluteOptions[VisualizeKnot[knot], VertexCoordinates], 0.1];
                                    shift = LocalSplittingDirections[knot];
                                    Show[ReproduceGraph[VisualizeKnot[knot]],
    Graphics3D[{Red, Arrow[{pos[[n]], pos[[n]] + shift[[n]]}]}]]
                            ];

DrawLocalSplit[t[4], 1]
```



```
Table[DrawLocalSplit[t[i]], {i, 5}]
```

{  ,  ,

 ,  ,  }

The function *FindNeighbors* outputs to any knot a list of lists of which the inner list at the *i*th position contains all vertices that are connected to the *i*th vertex, both in *canonical labels*. In addition it also outputs the association corresponding to this transformation of labels. The function *TestNeighbors* is simply a way to test if *FindNeighbors* is working properly. Since these functions act in a very trivial way on the trefoil, I'll use the knot *t*[2] to demonstrate these functions below.

```
FindNeighbors[knot_] := Module[{a, v, conn,
     relabel = Association[{}], j = 1, pairs, neighbors1, neighbors2, neighbors},
                                                         a = VisualizeKnot[knot];

   v = Round[ VertexCoordinates /. AbsoluteOptions[a, VertexCoordinates], 0.1];
                                                         conn = FindConnections[knot];

   Do[Do[If[! ContainsAll[Keys[relabel], {i[[k]]}],
      AssociateTo[relabel, i[[k]] → j]; j++;], {k, {1, 2}}], {i, conn}];
                                                         pairs = Take[#, 2] & /@ conn;
                                                         neighbors1 =
    Table[{i}~Join~List[DeleteCases[Table[If[MemberQ[p, i], If[First[p] == i,
           Last[p], First[p]]], {p, pairs}], Null]], {i, Keys[relabel]}];

   neighbors2 = neighbors1 /. relabel;

   {neighbors = Last[#] & /@ neighbors2, relabel}
                                    ];
```

```
FindNeighbors[t[2]][[1]]
```

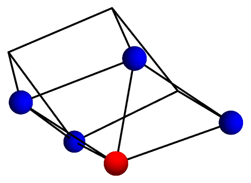
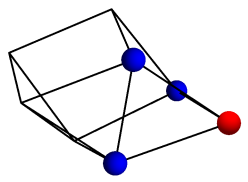
```
{{2, 3, 4, 5}, {1, 6, 5}, {1, 6, 4, 8},
 {1, 3, 5, 8}, {1, 2, 4, 7}, {2, 3, 7}, {6, 5, 8}, {3, 4, 7}}
```

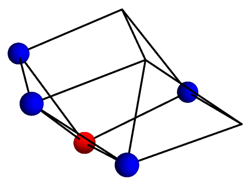
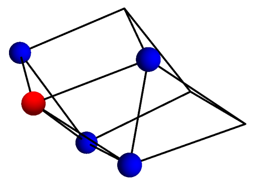
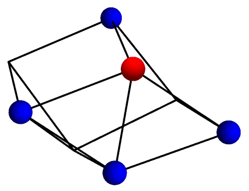
```
FindNeighbors[t[2]][[2]]
```

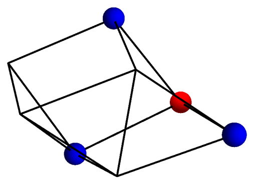
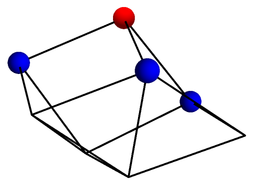
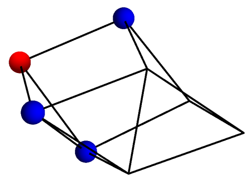
```
<|1 → 1, 7 → 2, 11 → 3, 5 → 4, 13 → 5, 3 → 6, 9 → 7, 15 → 8|>
```

```
TestNeighbors[knot_, n_] := Module[{a, v, conn, relabel = Association[{}],
    j = 1, pairs, neighbors1, neighbors2, neighbors, ϵ},
                                                  a = VisualizeKnot[knot];

   v = Round[ VertexCoordinates /. AbsoluteOptions[a, VertexCoordinates], 0.1];
                                                  conn = FindConnections[knot];

   Do[Do[If[! ContainsAll[Keys[relabel], {i[[k]]}],
      AssociateTo[relabel, i[[k]] → j]; j++;], {k, {1, 2}}], {i, conn}];
                                                  pairs = Take[#, 2] & /@ conn;
                                                  neighbors1 =
    Table[{i}~Join~List[DeleteCases[Table[If[MemberQ[p, i], If[First[p] == i,
          Last[p], First[p]]], {p, pairs}], Null]], {i, Keys[relabel]}];

   neighbors2 = neighbors1 /. relabel;

   neighbors = Last[#] & /@ neighbors2;
                                                  ϵ = 0.1;

   Show[ReproduceGraph[a], Graphics3D[    {Red, Ball[v[[n]], ϵ]   ,
      Blue, Table[Ball[v[[m]], ϵ] , {m, neighbors[[n]]}]    } ]     ]
                                           ];
```

```
Table[TestNeighbors[t[2], i], {i, Length[FindNeighbors[t[2]][[1]]]}]
```

{  ,  ,

 ,  ,  ,

 ,  ,  }

The careful reader will have noticed, that *LocalSplittingDirections* can only find the desired direction of spearation between lower and upper part of the vertices up to a sign. This cas also be seen in the plots above. This means before plotting the final output, we have to make sure that the direction of the splitting of a vertex, by convention I choose the vector to point from the lower to the upper part, agrees with the handedness of the vertex. To check this, the following function *ComputeIncomingVectors* computes first the direction of the edge representing the incoming lower arc of the corresponding crossing and second the direction of the edge represention the upper arc incoming to the crossing if the crossing is left handed (Xm) or outgoing if the crossing is right handed (Xp). So the output is nothing else than a list of the pairs of normalized vectors where each pair {vl, vu} is the pair of directions of lower and upper arc comming in or going out of the crossing. With this output the final function *FinallyPlotTheKnot* can check whether the splitting direction of a knot is correct and if it's not, it can change reverse the direction.
The interesting part about this function is the repeated transition between *natural* and *canonical labels.*

```
ReverseAssociation[a_] := Association[Table[a[i] → i, {i, Keys[a]}]];
ComputeIncomingVectors[knot_, v_, shift_] := Module[{rawData, n, a, or, por,
    result = {}, b, nat, c, d, vlpos, f, SecondEntry, g, vupos, vl, vu},
                                rawData =
    knot /. Times → List /. {Xm → List , Xp → List};
                                {n, a} = FindNeighbors[knot];
                                or = knot /. Times → List /.
      {Xp[i_, j_, k_, l_] :→ {i, "p"}, Xm[i_, j_, k_, l_] :→ {i, "m"}} /. a // Sort;
                                por = Last[#] & /@ or;
                                b = ReverseAssociation[a];
                                Do[
                                    nat = can /. b;
    (*Start with the canonical labels
     of each vertex and transform back into natural labels*)

    c = First @ First@ DeleteCases[Position[rawData, nat], {i_, 1}];
                                    d = rawData[[c, 1]];
                                    vlpos = d /. a;
    (*Find the canonical label of the
     vertex to which nat is not the incoming lower arc*)
                                    (*Use for this the fact
     that every number in the knot data appears exactly twice*)
                                    (*So vlpos is the vertex
     label of the vertex from which the incoming lower arc starts*)

    f = Position[rawData, nat] // SortBy[#[[2]] &] // First // First;
                                    SecondEntry = rawData[[f, 2]];
    (*Second entry of vertex where nat /.b is first*)

    g = DeleteCases[Position[rawData, SecondEntry], {f, 2}] // First // First;
                                    vupos = rawData[[g, 1]] /. a;
    (*Repeat the same process as above to find the label
     of the vertex from which the second incoming arc starts*)
                                    vl = v[[can]] - v[[vlpos]] // Normalize;
                                    vu = v[[can]] - v[[vupos]] // Normalize;
                                    AppendTo[result, {vl, vu}];
                                 , {can, Values[a]}];
                                result
                                    ];
```

```
knot = t[2];
a = VisualizeKnot[knot];
v = Round[ VertexCoordinates /. AbsoluteOptions[a, VertexCoordinates], 0.1];
shift = 0.2 * LocalSplittingDirections[knot];

ComputeIncomingVectors[t[2], v, shift] // Transpose // MatrixForm
```

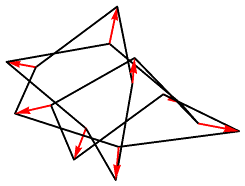
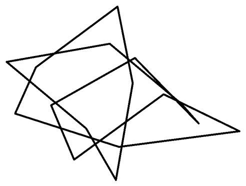
```
⎛⎛ 0.948683 ⎞ ⎛ 0.829561 ⎞ ⎛ 0.784465 ⎞ ⎛-0.226455⎞ ⎛-0.11547⎞ ⎛-0.704361⎞ ⎛-
⎜⎜ 0.       ⎟ ⎜ 0.20739  ⎟ ⎜-0.196116 ⎟ ⎜-0.566139⎟ ⎜ 0.57735⎟ ⎜ 0.616316⎟ ⎜
⎜⎝-0.316228 ⎠ ⎝-0.518476 ⎠ ⎝-0.588348 ⎠ ⎝ 0.792594⎠ ⎝ 0.80829⎠ ⎝-0.35218 ⎠ ⎝
⎜⎛ 0.737865 ⎞ ⎛ 0.693103 ⎞ ⎛-0.545455 ⎞ ⎛-0.948683⎞ ⎛-0.829561⎞ ⎛-0.704361⎞ ⎛
⎜⎜-0.527046 ⎟ ⎜ 0.693103 ⎟ ⎜-0.818182 ⎟ ⎜ 0.      ⎟ ⎜-0.20739 ⎟ ⎜ 0.616316⎟ ⎜
⎝⎝ 0.421637 ⎠ ⎝ 0.19803  ⎠ ⎝-0.181818 ⎠ ⎝ 0.316228⎠ ⎝ 0.518476⎠ ⎝-0.35218 ⎠ ⎝
```

Assembling All Parts

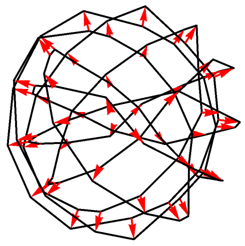
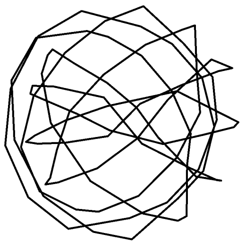
So, finally we can write a function that uses all of the work above. I'll now go over how it works in detail.
1) First it uses *VisualizeKnot* to compute the vertex coordinates of the crossings of the knot in the variable *v* where the coordinate at the i^{th} position belongs to the i^{th} vertex in *canonical labels*.
2) It then lets the function *LocalSplittingDirections* compute the shifts of each vertex and creates two lists of coordinates, one for the upper and one for the lower vertices, both corresponding to *canonical labels.*
3) With the information provided by *ComputeIncomingVectors* the first *Do loop* checks each of the splitting direction, meaning it checks if they agree with the handedness. This is done by checking if the cross product of *vl* and *vu* aligns with the splitting direction. In other words, check if the volume of the parallelepiped spanned by the vectors {*vl,vu,shift*} has positive volume. Compare for example the arrows attached to the knot *t[5]* (the ball) in the the plot above and below.
4) The information how the splitted vertices connect and how they are relabeled is provided by *CreateS-plitAdjacency.* The second *Do loop* then sorts the coordinates of the vertices in *vup* and *vlow* which correspond to *canonical labels* to the order in which the upper and lower vertices appear after splitting, which is how they appear in *adj*, the split adjacency matrix. This gives us a list *vsorted* with the coordi-nates in the right order of *split labels*.
5) Finally it is just a matter of reusing code from the function *ReproduceGraph* to create the graphics object representing the final knot!!

```
FinallyPlotTheKnot[knot_, arrows_] := Module[{a, v, vup, vlow, shift, income,
    adj, ass, done = Association[{}], vsorted = {}, j = 1, d, e, f, g},
                                a = VisualizeKnot[knot];

   v = Round[ VertexCoordinates /. AbsoluteOptions[a, VertexCoordinates], 0.1];
                                (*vlow= # - {0,0,.2}&/@v;
                                vup = # + {0,0,.2}&/@v;*)
                                shift = 0.2 * LocalSplittingDirections[knot];
                                vlow = v - shift;
                                vup = v + shift;
                                income = ComputeIncomingVectors[knot, v, shift];
                                Do[
                                    If[

     Negative[Cross[income[[z, 1]], income[[z, 2]]].shift[[z]]],

     {vup[[z]], vlow[[z]]} = {vlow[[z]], vup[[z]]}
                                          ];
                                 , {z, Length[income]}];

                                {adj, ass} = CreateSplitAdjacency[knot];
                                Do[

    If[! ContainsAll[Keys[done], {Abs[i]}], AssociateTo[done, Abs[i] → j]; j++;];
                                    If[Positive[i], AppendTo[vsorted,
     vup[[done[Abs[i]]]]], AppendTo[vsorted, vlow[[done[Abs[i]]]]] ];
                                 , {i, Keys[ass]}];
                                d = Dimensions[adj][[1]];
                                e = Flatten[Table[If[adj[[i, j]] == 1,
      Line[List[vsorted[[i]], vsorted[[j]]]], {}], {i, d}, {j, d}], 1];
                                f = DeleteCases[e, {}];

   g = Table[Arrow[{vlow[[i]], vup[[i]]}], {i, Length[v]}];
                                Graphics3D[
    f~Join~{Red}~Join~If[arrows == "on", g, {}], Boxed → False]
                                ];
```

```
Table[FinallyPlotTheKnot[t[2], arrows], {arrows, {"on", "off"}}]
```

{  ,  }

```
Table[FinallyPlotTheKnot[t[5], arrows], {arrows, {"on", "off"}}]
```

{  ,  }

I have to admit that this project is far from being finished. This is because there are a lot of details that could be improved. To name only two of them, I would like the line in the final plot representing the knot to have no kinks and also some physical expansioan as for example a tube.
Moreover is this process very fragile and tends to errors when it comes to two vertices sharing more than one edge and the same splitting direction, because then the lines connecting upper and lower part may either cross or be parallel. Both can’t happen, since we have by constriction taken care of every crossing, so there can’t be any more crossings. Moreover if they are parallel, the crossings cancel each other leaving less crossings than we need to represent the right knot. The first issue can be addressed by considering in this special case the plane that is uniquely determined by the parallel splitting directions of the vertices and bulging both lines in the direction orthogonal to this plane in the one way out of two possible ones such that the handedness of all crossings is preserved. For the second problem, I have no idea of how to fix it!

```
FinallyPlotTheKnot[t[1]]
```

```
FinallyPlotTheKnot[Xp[1, 5, 2, 4] Xp[3, 1, 4, 6] Xp[5, 3, 6, 2]]
```

```
? :>
```

lhs :> *rhs* or *lhs* :→ *rhs* represents a rule that transforms *lhs* to *rhs*, evaluating *rhs* only after the rule is used. >>