Exercise 7: Loop subdivision

Implement the Loop subdivision scheme as a function of the `TriangleMesh` data structure. Following steps should be done:

1) New array of vertices should be allocated. You can assume that the mesh is closed and genus 0. In that case, the number of new vertices is equal to V + F*3/2, where V is the number of vertices and F is the number of triangles.
2) The first V vertices should be derived from the original vertices, using the rule for old vertices. The weights for the neighboring vertices in the Loop scheme is 1, while the weight for the central vertex depends on its degree n:

$$w = \frac{64n}{40 - (3 + 2\cos\left(\frac{2\pi}{n}\right))^2} - n$$

3) For each edge, a new vertex should be generated. There are many ways to achieve that using particular data structure at hand. If the data structure at hand is a corner table, then it is a good procedure to generate a vertex for each corner in the sense that the vertex is generated for the edge that lies across the corner. The corners are numbered 0-(3F-1), thus it is not difficult to iterate through them. For each corner, it is possible to identify the next and previous corner and their respective vertices. Traversing the mesh further, it is possible to obtain all the vertices needed for the new vertex rule of the Loop scheme. Note that an edge corresponds to two corners, and it is therefore necessary to only generate the new vertex for one of them. Possible solution is to generate the new vertex only if (corner index) < opposite[corner index]. Also, it is important to save the index of each newly generated vertex, so that it can be used in the following step. A good way to do that is to allocate an array of indices `indices[3F]`, one for each corner.
4) New set of triangles must be created. Again, the concrete procedure depends on the used data structure. In the previously described case, four new triangles are generated for each old triangle, using the indices of the old triangle and the indices of vertices lying at the edges, stored in the `indices[]` array in previous step.

Task 1: implement the connectivity subdivision procedure

Below is the source code using the CornerTable data structure. Either adapt the code for your data structure, or directly paste it into the TriangleMesh source file. Use empty calls `oldPoint()` and `newPoint()`, these will be implemented later.

```
public TriangleMesh Loop()
        {
            TriangleMesh result = new TriangleMesh();
            result.points = new Point3D[this.points.Length+this.triangles.Length*3/2]; // this
will not work for meshes with border or non-zero genus!

            // first take care of the old points
            for (int i = 0; i < points.Length; i++)
            {
                result.points[i] = oldPoint(i);
            }

            // now the new points, one for each edge
            int index = points.Length;
            int[] indices = new int[opposite.Length];
            for (int i = 0; i < opposite.Length; i++)
            {
                // an edge lies opposite to two corners, make sure we pick just one of them
                if (i<opposite[i])
                {
                    // we ask for a position of a new vertex, which lies on the edge opposite 
to the given corner
                    result.points[index] = newPoint(i);
                    indices[i] = index;
                    indices[opposite[i]] = index;
                    index++;
                }
            }

            // now we have to fill the new connectivity
            // four triangles for each old triangle (dyadic split)
            result.triangles = new Triangle[this.triangles.Length * 4];
            for (int i = 0; i < triangles.Length; i++)
            {
                Triangle t = triangles[i];

                // these are the corners
                int c1 = 3 * i;
                int c2 = 3 * i + 1;
                int c3 = 3 * i + 2;

                // these are the new vertices that lie on the edges
                int e1 = indices[c1];
                int e2 = indices[c2];
                int e3 = indices[c3];

                // now just construct the new triangles
                result.triangles[4 * i] = new Triangle(t.V1, e3, e2);
                result.triangles[4 * i + 1] = new Triangle(t.V3, e2, e1);
                result.triangles[4 * i + 2] = new Triangle(t.V2, e1, e3);
                result.triangles[4 * i + 3] = new Triangle(e1, e2, e3);
            }

            // thats all folks.
            return (result);
        }
```

Task 2: implement the newVertex and oldVertex callbacks (2 points)

Implement the two functions `newVertex` and `oldVertex`, which should return the correct positions for a new vertex added to an edge that lies opposite to a corner, and an updated position of an old vertex. Use the formula above to compute it. Consider how you would have to change the methods in order to implement the Butterfly scheme.

Task 3: Snap the vertices to their limit position (3 points)

Remember that it is possible to locally represent the subdivision by the subdivision matrix S. We wish to find a vector of limit points $\mathbf{p}^* = S^\infty \mathbf{p}$. In order to do that, we find the eigenvectors of S, called v_i, and stack them as columns of a matrix $V = [v_0, v_1, \dots, v_n]$. Now, we can rewrite $\mathbf{p}^* = S^\infty \mathbf{p} = S^\infty I \mathbf{p} = S^\infty V V^{-1} \mathbf{p}$. Note that the eigenvectors are ordered according to the eigenvalues, and we assume that the first eigenvalue is equal to one, while the remaining ones smaller than 1 in magnitude. Thus $S^\infty V = [v_0, 0, \dots, 0]$. Therefore we have to compute the first row of V^{-1}, denoted b_0, and this allows us to compute $p_0^* = v_0^0 b_0^0 p_0 + v_0^0 b_0^1 p_1 + \dots + v_0^0 b_0^n p_n = \sum_i w_i p_i$. Therefore, what we need to compute are the weights w_i for each possible vertex degree.

We will therefore build a function `double[] snapWeights(int n)` that returns a set of weights for each possible vertex degree n. This function will then be used to obtain the weights on demand while snapping the vertices to their limit positions.
In order to perform the eigendecomposition in the `snapWeights` function, we are going to use the `mathnet numerics` library. First, you should construct the subdivision matrix S, as an instance of the `Matrix` class. Filling the matrix of course depends on the vertex degree n.

Then, we will perform the eigendecomposition of the matrix. In order to do that, use the `Evd()` method of the `Matrix<double>` class. It returns an instance of the `Evd<double>` class, which encapsulates the decomposition. Most importantly, it provides the `EigenVectors` property, which is a matrix where the eigenvectors are stacked as columns, i.e. it is exactly the V matrix that we need for our computation. By calling the `Inverse()` function on it, we can also obtain its inverse. Out of this inverse, we actually only need the first row, b_0. The weights needed for the snapping are then $w_i = v_0^0 b_0^i$. If they are computed correctly, they should sum up to 1.

Finally, in the `SnapLoop()` function, you should compute the limit position for each vertex. To do that, you should allocate a new array of vertices, and for each vertex, use the weights, obtained from the `snapWeights` function, to compute the limit position. If you wish to improve the efficiency, you can store the weights in a dictionary, so that they can be reused for other vertices of the same degree.