GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN

maniflow

A Python library for geometric processing

Felix Widmaier, Robin Mark Riegraf, Yangshan Xiang, and Minming Zhao

Georg-August-Universität Göttingen

Abstract

The purpose of this library is to provide tools for the study of the most beautiful discipline of mathematics: geometry and geometric analysis. In doing so, we restrict ourselves to 2-manifolds. For a mathematician, this may initially seem like a major restriction. However, it allows us, in a relatively simple way, to represent 2-manifolds as "meshes" and to develop powerful tools to study them.

The abstraction hardly needs to be restricted at all, because the proposed calculus for meshes makes it possible to develop new geometries with comparatively little effort. Properties of these meshes can then be examined using `maniflow`. For example, we provide tools to break down meshes into their connected components. You can also use `maniflow` to determine the orientability of a mesh. It is also possible to run a geometric flow, such as the mean curvature flow, on a mesh. This means that `maniflow` can also be used to examine meshes with regard to curvature (Gaussian curvature, mean curvature).

`maniflow` also provides the option of creating images of the meshes. This makes it possible, for example, to create animations of geometric flows etc.

Contents

List of Figures

Göttingen, March 18, 2024

`maniflow` was developed as part of the course M.Mat.0731 "Advanced practical course in scientific computing" at Georg-August University Göttingen.

The image on the frontpage is taken from `https://unsplash.com/de/fotos/blaues-und-rotes-licht-digitales-hintergrundbild-wxj729MaPRY`

Getting started

> “Be patient, for the world is broad and wide.“
>
> – E. A. Abbott, Flatland: A Romance of Many Dimensions

The code of `maniflow` was originally published on

`https://gitlab.gwdg.de/yangshan.xiang/scientific-computing`

To install the libary, simply use

```
pip install dist/maniflow-1.0-py2.py3-none-any.whl
```

To build the wheel file of the library, use

```
python setup.py bdist_wheel -universal
```

Dependencies. The installation and usage of `maniflow` requires the following packages to be installed: `numpy`, `pillow`

Optional dependencies. When using `maniflow.render.SVGPainterRenderer`, one requires the installation of `drawsvg`.

1 Introduction

First, we will look at the basic mathematical concepts that ultimately underpin the whole theory. So let's start with so-called meshes and look at some examples and how these mathematical concepts can be implemented in code using `maniflow`.

Definition 1 (Mesh)**.** *Let V be a vector space over $\mathbb{R}$ of dimension n. Let $\mathcal{V}_M \subset V$ be a set of points in V. We further let $\mathcal{F}_M \subset \mathcal{V}_M^3$. The pair $M = (\mathcal{V}_M, \mathcal{F}_M)$ is then called mesh. The elements of $\mathcal{V}_M$ are called points of M and the elements of $\mathcal{F}_M$ are the faces of the mesh M.*

For a mesh $M = (\mathcal{V}_M, \mathcal{F}_M)$ we will often denote $V_M = |\mathcal{V}_M|$ and $F_M = |\mathcal{F}_M|$.

Remark. Meshes M can be considered as 2-dimensional simplicial complexes. Thus for 2-dimensional manifolds $\tilde{M} \subset V$ we may find a *triangulation* simplicial complex K of $\tilde{M}$. The corresponding mesh will be called *triangulation* mesh of the manifold $\tilde{M}$.

Example 1 (Tetrahedron). *Let*

$$\mathcal{V} = \left\{ \left(\sqrt{\frac{8}{9}}, 0, -\frac{1}{3}\right), \left(-\sqrt{\frac{2}{9}}, \sqrt{\frac{2}{3}}, -\frac{1}{3}\right), \left(-\sqrt{\frac{2}{9}}, -\sqrt{\frac{2}{3}}, -\frac{1}{3}\right), (0,0,1) \right\} \subset \mathbb{R}^3$$

and $\mathcal{F} = \{f \in 2^{\mathcal{V}} : |f| = 3\}$. *The mesh* $T = (\mathcal{V}, \mathcal{F})$ *is the tetrahedron, which is displayed in figure 1. This*

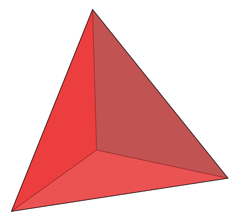


Figure 1: Tetrahedron

can be implemented using `maniflow` *by using the* `Mesh` *class:*

```
import numpy as np
import itertools
from maniflow.mesh import Mesh, Face

# computing the four vertices of the tetrahedron
v1 = np.array([np.sqrt(8/9), 0, -1/3])
v2 = np.array([-np.sqrt(2/9), np.sqrt(2/3), -1/3])
v3 = np.array([-np.sqrt(2/9), -np.sqrt(2/3), -1/3])
v4 = np.array([0, 0, 1])

tetra = Mesh()
# setting the vertices as the vertices of the new mesh object
tetra.vertices = [v1, v2, v3, v4]
# now we compute the subsets of all the vertices consiting of three vertices
subsets = set(itertools.combinations(list(range(tetra.v)), 3))
# the faces are then set as the faces of tetra
tetra.faces = [Face(tetra, *list(i)) for i in subsets]
```

This way, we obtain the `Mesh` *object* `tetra` *which represents a tetrahedron.*

1.1 Reading and writing `.obj` files

The repeated computation of meshes can require a lot of computing capacity under certain circumstances. It can also be difficult to programme complicated geometries line by line in the code. To avoid these difficulties, `maniflow` supports the `.obj` file format. This makes it possible to export meshes for further editing in other programmes (e.g. Blender or similar) or to display them[1]. You can also use third-party software to create complicated geometries relatively easily and import them into `maniflow`. Interaction with .obj files is made possible by the maniflow.mesh.obj.OBJFile class.

[1] if you do not want to use the internal renderer of `maniflow`

Caution: `maniflow.mesh.obj.OBJFile` currently disregards any normal vectors, texture coordinates, line elements etc. defined in the `.obj` file. Only the coordinates of the vertices are taken into account and for the faces only the indices of the vertices that make them up are taken into consideration.

Example 2. *Consider the `Mesh` object `tetra` from example 1. To store this mesh in a `.obj` file, we may use*

```
from maniflow.mesh.obj import OBJFile

OBJFile.write(tetra, "examples/tetrahedron.obj")
```

This code produces the file `examples/tetrahedron.obj`. In order to load a mesh into `maniflow`, simply use

```
mesh = OBJFile.read("examples/tetrahedron.obj")
```

1.2 `maniflow.mesh.utils.VertexFunction` – Creating meshes from parameterisations

The way we created a mesh of a tetrahedron in the previous example is very static and absolutely not suitable if you want to study more complicated geometries. `maniflow`, however, provides the option of creating meshes quite easily using parameterisations. For this purpose, `maniflow` provides the wrapper `maniflow.mesh.utils.VertexFunction`, which executes a given function on all vertices of the mesh and has the resulting mesh as output.

Example 3. *For the following example, we assume that we have a `Mesh`-object `mesh` and we want to shift this mesh by the vector $\begin{pmatrix}1 & 2 & 3\end{pmatrix}^\intercal \in \mathbb{R}^3$. For this we make use of a `VertexFunction`:*

```
# importing the wrapper from maniflow.mesh.utils
from maniflow.mesh.utils import VertexFunction

# implementing the VertexFuntion 'shift'
@VertexFuntion
def shift(vertex):
    return vertex + np.array([1, 2, 3])


# applying 'shift' to 'mesh'
shifted = shift(mesh)
```

The resulting `Mesh`, `shifted`, is `mesh` shifted by the vector $\begin{pmatrix}1 & 2 & 3\end{pmatrix}^\intercal \in \mathbb{R}^3$.

Another application of this would be the creation of meshes from parameterisations $\psi\colon \mathbb{R}^2 \supset D \to \mathbb{R}^3$. Oftentimes, the domain D is a cartesian product of two intervals, so $D = I_1 \times I_2$.[2] For this, `maniflow` provides the class `maniflow.mesh.parameterized.Grid`.

Example 4 (Moebius strip)**.** *We now turn to an example where we want to create a triangulation of a moebius strip. To this end, we will use the parametrisation*

$$\psi\colon [0, 2\pi] \times [-1, 1] \to \mathbb{R}^3,\ (u, v) \mapsto \begin{pmatrix} \left(1 + \frac{v}{2}\cos\frac{u}{2}\right)\cos u \\ \left(1 + \frac{v}{2}\cos\frac{u}{2}\right)\sin u \\ \frac{v}{2}\sin\frac{u}{2} \end{pmatrix}.$$

[2] Or to be more precise, `maniflow` makes it easy to create meshes from parametrisations, where the domain D is homoeomorphic to a square.

In order to discretise the set $[0, 2\pi] \times [-1, 1]$, we make use of `maniflow.mesh.parametrized.Grid` *in order to create a high resolution lattice. Then we can implement a* `VertexFunction` *to capture the parametrisation ψ in code and apply it to our lattice.*

```
from maniflow.mesh.parameterized import Grid
from maniflow.mesh.utils import VertexFunction


# implementing the parametrisation 1:1 in code as a VertexFunction
@VertexFunction
def moebius(vertex):
    x = vertex[0]
    y = vertex[1]
    x0 = np.cos(x) * (1 + (y / 2) * np.cos(x / 2))
    x1 = np.sin(x) * (1 + (y / 2) * np.cos(x / 2))
    x2 = (y / 2) * np.sin(x / 2)
    return np.array([x0, x1, x2])


u = Grid((0, 2 * np.pi), (-1, 1), 30, 10)  # create a high resolution grid
moebiusMesh = moebius(u)  # mapping the vertices from the grid according to the parametrisation
coincidingVertices(moebiusMesh) # remove the redundant vertices at the joint after making the moebius band
```

With this we obtain the `Mesh`*-object* `moebiusMesh`*. Using* `maniflow.mesh.obj.OBJFile`*, we can write this mesh to memory as a* `.obj` *file, see section 1.1*

```
from maniflow.mesh.obj import OBJFile

OBJFile.write(moebiusMesh, "examples/moebius.obj")
```

Unsurprisingly, one can then load this file into Blender and create pictures of it etc.[3] *Figure 2 shows a screenshot taken from Blender with the* `.obj` *file from* `examples/moebius.obj`*.*

[3] `maniflow` comes with its own simple renderer. But if you want to do more elaborated computer graphics, you might consider using some other software to render images.

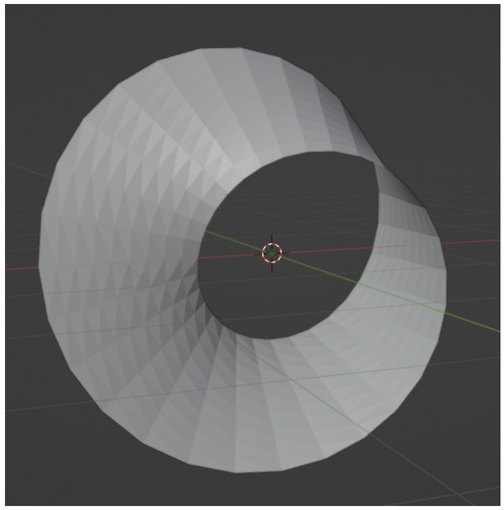


Figure 2: Screenshot of Blender with a moebius strip made with `maniflow`

2 The face graph of a mesh

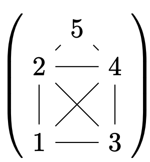
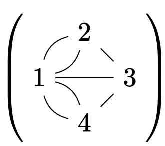
Definition 2 (Undirected Graph)**.** *Let $\mathcal{V}_G$ be a set and $\mathcal{E}_G \subset \{e \in 2^{\mathcal{V}_G} : |e| = 2\}$ be a set of unordered pairs of elements from $\mathcal{V}_G$. The pair $G = (\mathcal{V}_G, \mathcal{E}_G)$ is then called undirected Graph. The elements from $\mathcal{V}_G$ are called vertices of G and the elements from $\mathcal{E}_G$ are called edges of G.*

For a Graph $G = (\mathcal{V}_G, \mathcal{E}_G)$ we write

$$x \;—\; y$$

if $\{x, y\} \in \mathcal{E}_G$. If we take all edges and points together in this way, we get the picture of a graph with undirected edges.

Example 5.

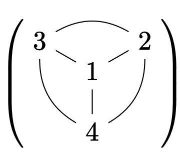
G:  , H:  (1)

Definition 3 (Face Graph)**.** *Let $M = (\mathcal{V}_M, \mathcal{F}_M)$ be a mesh and*

$$\mathcal{E} = \big\{(f_1, f_2) \in \mathcal{F}_G^2 : |f_1 \cap f_2| = 2\big\}$$

The face graph of M is the graph $(\mathcal{F}_M, \mathcal{E})$.

Example 6. *The face graph of the tetrahedron is given by*

G:  (2)

Algorithm 1: Construction of the face graph of a given mesh

Input : A mesh $M = (\mathcal{V}_M, \mathcal{F}_M = \{f_1, f_2, f_3 \ldots\})$
Output: The adjacency matrix of the face graph of the mesh M

$G := 0 \in \mathbb{R}^{F_M \times F_M}$;
for $i = 1$ **to** F_M **do**
    neighbors $:= 0$;
    for $j = 1$ **to** F_M **do**
        if *neighbors* $= 3$ **then**
            break;
        end
        if $|f_i \cap f_j| = 2$ **and** $i \neq j$ **then**
            $G_{ij} \leftarrow 1$;
            neighbors $\leftarrow$ *neighbors* $+ 1$;
        end
    end
end
return G

The face graph of a given mesh can be constructed by algorithm 1. Since this algorithm loops over the faces of the mesh in a nested way, the complexity of it lies in $O(F_M^2)$. As this runtime complexity has the consequence of the algorithm being very slow at execution for somewhat large meshes, the face graph is computed dynamically by `maniflow.mesh.Mesh.faceGraph`.

2.1 A first application: `maniflow.mesh.utils.connectedComponents`

The method `maniflow.mesh.utils.connectedComponents` decomposes the given mesh into its connected components. Now that we have an algorithm with which to compute the face graph, the connected components of a mesh can now be identified as the connected components of the face graph. These can be determined via the breadth-first traversal of the face graph.

Algorithm 2: Construction of the face graph of a given mesh

Input : A mesh $M = (\mathcal{V}_M, \mathcal{F}_M = \{f_1, f_2, f_3 \ldots\})$
Output: The connected components of the mesh M

Compute the adjacency matrix G using 1;
start $:= 1$;
$n := 1$;
while $\mathcal{F}_M \neq \emptyset$ **do**
    Compute a breadth first traversal sequence $T_n \leftarrow \{f_{start}, f_b, f_c, \ldots\} \subseteq \mathcal{F}_M$;
    $n \leftarrow n + 1$;
    $\mathcal{F}_M \leftarrow \mathcal{F}_M \setminus T_n$;
    Set $1 < start \leq F_M$ such that $f_{start} \in \mathcal{F}_M$;
end
return T_1, T_2, $\ldots$

Runtime analysis. Algorithm 1 has a runtime complexity which lies in $O(F_M^2)$. The breadth-first traversal on the face graph has a runtime[4] complexity of $O(F_M + 3 \cdot F_M) = O(F_M)$. The computation of $\mathcal{F}_M \setminus T_n$ has also quadratic complexity $O(|\mathcal{F}_M|^2)$. Thus the overall complexity of algorithm 2 lies in $O(F_G^2)$.

Example 7. *In this example we analyse the connected components of the teapot from* `examples/teapot.obj`*. The teapot is displayed in figure 3.*

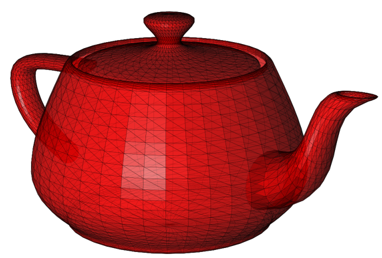


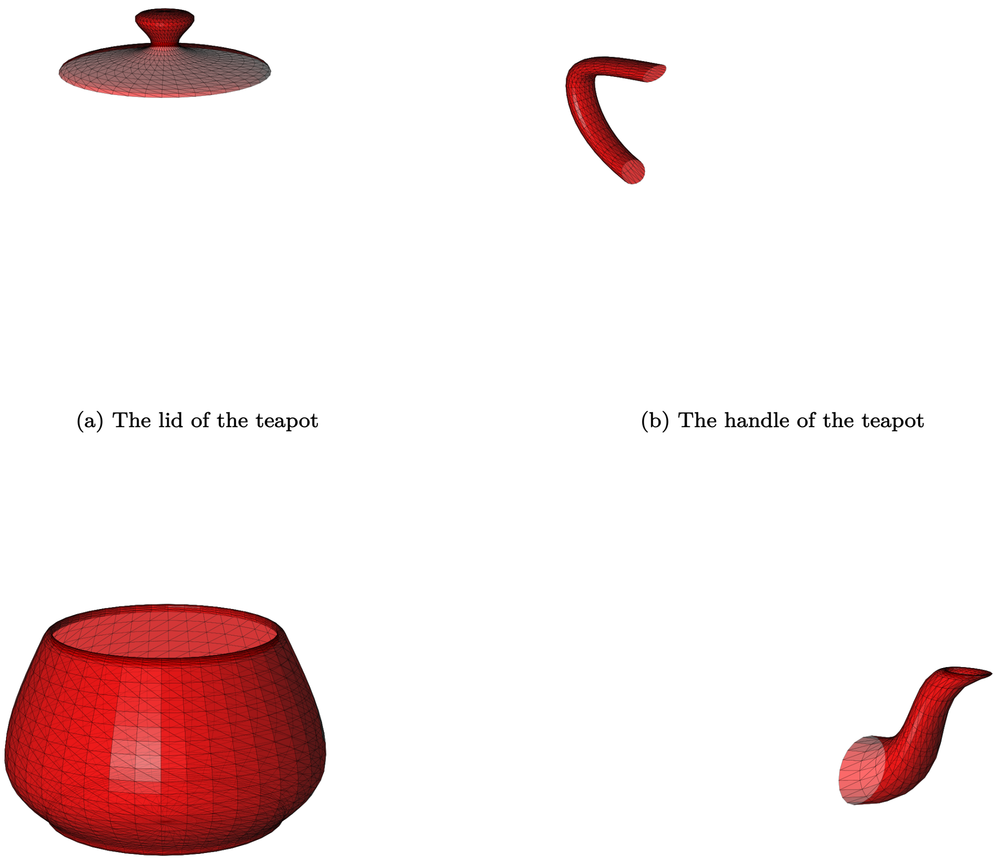
Figure 3: The teapot from `examples/teapot.obj`

The connected components can be computed using the following code

```
from maniflow.mesh import Mesh
from maniflow.mesh.obj import OBJFile
from maniflow.mesh.utils import connectedComponents, coincidingVertices

teapot = OBJFile.read("examples/teapot.obj")  # reading the mesh from memory
coincidingVertices(teapot)  # identifying and collapsing coinciding vertices

# now we compute the connected components
components = connectedComponents(teapot)

# we can now reconstruct meshes from these lists of faces and write them to .obj
    files
for i, component_list in enumerate(components):
    component = Mesh.fromFaceList(teapot, *component_list)
    OBJFile.write(component, "teapot" + str(i + 1) + ".obj")
```

The resulting components are shown in figure 4.

[4] Since on a graph with the number of vertices being V and the number of edges being E the breadth first search has a complexity of $O(E + V)$. As every face has at most three neighbors we obtain the given runtime complexity.



(a) The lid of the teapot

(b) The handle of the teapot

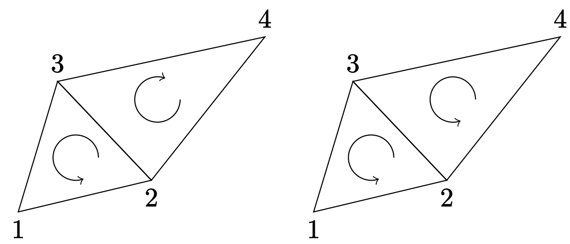
(c) The body of the teapot

(d) The spout of the teapot

Figure 4: The connected components of the teapot

2.2 Another application: checking orientability of a `Mesh`

Orientability is an important property of manifolds in geometry. For example, the area can only be meaningfully defined for orientable 2-manifolds. For 2-manifolds embedded in $\mathbb{R}^3$, orientability is equivalent to the existence of a continuous unit-normal vector field on the manifold. Similarly, we can extend the notion of orientability to meshes. If we consider a face spanned by the vertices $v_1,\ v_2$ and v_3, the normal vector is,



(a) Non compatible orientation (b) Compatible orientation

Figure 5: Two triangles with compatible and non compatible orientations

for example given by[5]

$$n = (v_1 - v_2) \times (v_1 - v_3).$$

Hence, the orientation or direction of n is dependent upon the enumeration of the vertices that define the face. A priori it is not clear that two neighboring faces really do have "compatible" enumerations of their vertices. So it may happen that two neighboring faces have normal vectors that point in "opposite" directions.[6]

Definition 4 (Orientation of faces)**.** *Let $f = (v_1, v_2, v_3)$ be a face defined by the vertices v_i for $1 \leq i \leq 3$. Let f' be another face that is defined by the vertices v_1, v_2 and v_*. We say that two faces have the same orientation, if there is a cyclic permutation $\sigma \in S_3$ such that*

$$(f_1, f_2) = (f'_{\sigma(2)}, f'_{\sigma(1)}).$$

Where f_i and f'_i denote the i-th vertex in the faces f and f' respectively.

In other words, two faces have the same orientation if they share an edge and that edge is traversed in opposite directions, when listing the vertices of the faces, see figure 5.

Example 8. *Consider the faces $f = (1, 2, 3)$ and $f' = (2, 3, 4)$. Then f and f' share the edge made up of the vertices 2 and 3. But both of them traverse this edge in the same direction $(2, 3)$. On the other hand, if we changed the enumeration of f' to $(3, 2, 4)$, the faces would have matching orientation. This is exactly the situation depicted in figure 5.*

In order to determine whether a given mesh is orientable or not, we can sort of "push" an orientation to the mesh by traversing each of the faces in breadth-first traversal and adjusting the orientations / enumerations of vertices of each of the neighboring faces for the currently traversed face.[7] This is what algorithm 3 does. The method `maniflow.mesh.utils.pushOrientation` implements this algorithm in `maniflow`. Finally, to check whether a mesh is orientable or not, we only need to iterate over all surfaces in the mesh and ask whether all neighbours have a compatible orientation. This is implemented as the method `maniflow.mesh.utils.isOrientable`.

[5]Note that n does not necessarily have length 1. But one may always obtain a unit normal vector by rescaling

[6]An appeal to the reader's intuition. You have to be careful with these notions.

[7]Of course we have to remember the faces we already considered while traversing the mesh in order to avoid faces being re-oriented twice.

Algorithm 3: Pushing an orientation to a mesh

Input : A mesh $M = (\mathcal{V}_M, \mathcal{F}_M = \{f_1, f_2, f_3 \ldots\})$

Store the connected components of M in $\mathcal{C}$ (see algorithm 2);
for $c \in \mathcal{C}$ **do**
    $visited := \emptyset$;
    for $f \in c$ **do**
        if $f \notin visited$ **then**
            Store neighbouring faces to f in $neighbours$;
            for $f_n \in neighbours$ **do**
                if *f and f_n do not have compatible orientations* **then**
                    reverse the enumeration of vertices in f_n;
                end
            end
            $visited \leftarrow visited \cup \{f\}$;
        end
    end
end

Runtime analysis. Algorithm 3 has a complexity of $O(F_M^2)$.

Example 9 (Orientability of the Moebius strip)**.** *Consider the* `Mesh` *object* `moebiusMesh` *from example 4. To determine whether this mesh is orientable or not, use*

```
from maniflow.mesh.utils import isOrientable

print(isOrientable(moebiusMesh))
```

As expected, the output will be `False` *as the Moebius strip is not orientable.*

Another example that is less trivial than the Moebius strip is documented in the file `examples/roman_surface.ipynb` where the so-called Roman surface is discussed.

3 Rendering meshes

`maniflow` employs rasterization, a fundamental technique in computer graphics, to render meshes. This process involves projecting each face of the mesh onto the viewing plane. The `maniflow.render.camera.Camera` class encapsulates the necessary matrices and operations for this projection.

`maniflow` provides three renderers whose functionality is basically the same. Firstly, the vertices of each surface of the mesh are projected onto the display plane by means of said projections. These projected polygons (triangles) are then drawn. The only difference between the three renderers provided is how the triangles are drawn.

3.1 The camera system

Standard OpelGL matrices... [1]

4 Geometry

4.1 Curvature

Example 10 (Gauss-Bonnet Theorem). *For a compact* 2*-manifold with* $\partial M = \emptyset$ *we have*

$$\int_M K \mathrm{d}A = 2\pi\chi(M).$$

Analogously we can state a discretized version:

$$\sum_{i \in \mathrm{vert}(M)} K_i = 2\pi\chi(M).$$

References

[1] Matthias Teschner. “Projections and Transformations in OpenGL”. In: *Image Processing and Computer Graphics* (2016). URL: https://www.cs.princeton.edu/courses/archive/spring22/cos426/72f0711e207865b0d6e5193b1f6d1f9b/PerspectiveProjection.pdf.