maniflow

A Python library for geometric processing

Felix Widmaier, Mark Robin 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, August 29, 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*

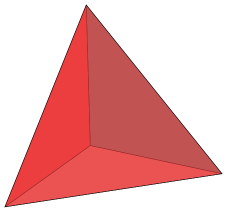


Figure 1: Tetrahedron

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 `manif low.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.

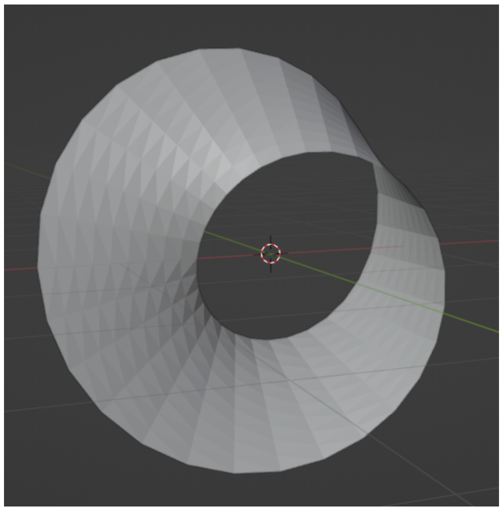


Figure 2: Screenshot of Blender with a moebius strip made with `maniflow`

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

$$x \longrightarrow y$$

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.

$$G\colon \left(\begin{tikzcd} & 5 \arrow[dl, dash] \arrow[dr, dash] & \\ 2 \arrow[rr, dash] \arrow[d, dash] \arrow[drr, dash] & & 4 \arrow[d, dash] \arrow[dll, dash] \\ 1 \arrow[rr, dash] & & 3 \end{tikzcd}\right), \qquad H\colon \left(\begin{tikzcd} & 2 \arrow[dr, dash] & \\ 1 \arrow[ur, dash, bend left] \arrow[ur, dash] \arrow[rr, dash, shift left] \arrow[rr, dash, shift right] \arrow[dr, dash, bend right] \arrow[dr, dash] & & 3 \\ & 4 \arrow[ur, dash] & \end{tikzcd}\right) \tag{1}$$

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*

$$G\colon \left(\begin{tikzcd} 3 \arrow[rr, dash, bend left] \arrow[dr, dash] \arrow[ddr, dash, bend right] & & 2 \arrow[dl, dash] \arrow[ddl, dash, bend left] \\ & 1 \arrow[d, dash] & \\ & 4 & \end{tikzcd}\right) \tag{2}$$

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 $M$
1  $G := 0 \in \mathbb{R}^{F_M \times F_M}$;
2  for $i = 1$ to $F_M$ do
3      $neighbors := 0$;
4      for $j = 1$ to $F_M$ do
5          if $neighbors = 3$ then
6              break;
7          end
8          if $|f_i \cap f_j| = 2$ and $i \neq j$ then
9              $G_{ij} \leftarrow 1$;
10             $neighbors \leftarrow neighbors + 1$;
11         end
12     end
13 end
14 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 $M$
1  Compute the adjacency matrix $G$ using 1;
2  $start := 1$;
3  $n := 1$;
4  while $\mathcal{F}_M \neq \emptyset$ do
5      Compute a breadth first traversal sequence $T_n \leftarrow \{f_{start}, f_b, f_c, \ldots\} \subseteq \mathcal{F}_M$;
6      $n \leftarrow n + 1$;
7      $\mathcal{F}_M \leftarrow \mathcal{F}_M \setminus T_n$;
8      Set $1 < start \leq F_M$ such that $f_{start} \in \mathcal{F}_M$;
9  end
10 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.*

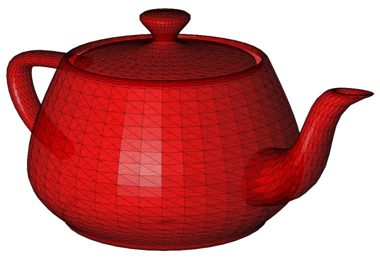


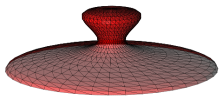
Figure 3: The teapot from `examples/teapot.obj`

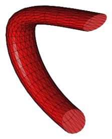
The connected components can be computed using the following code

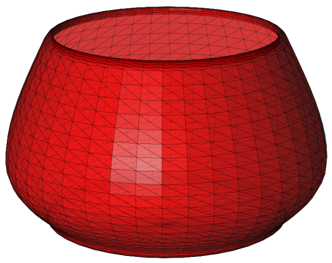
```
from maniflow.mesh import Mesh
from maniflow.mesh.obj import OBJFile
from maniflow.mesh.utils import connectedComponents, coincidingVertices

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

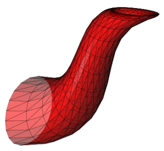
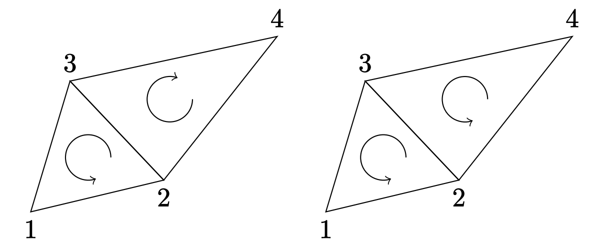

(d) The spout of the teapot

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 $M = (\mathcal{V}_M, \mathcal{F}_M = \{f_1, f_2, f_3 \ldots\})$
1  Store the connected components of $M$ in $\mathcal{C}$ (see algorithm 2);
2  for $c \in \mathcal{C}$ do
3      $visited := \emptyset$;
4      for $f \in c$ do
5          if $f \notin visited$ then
6              Store neighbouring faces to $f$ in $neighbours$;
7              for $f_n \in neighbours$ do
8                  if $f$ and $f_n$ do not have compatible orientations then
9                      reverse the enumeration of vertices in $f_n$;
10                 end
11             end
12             $visited \leftarrow visited \cup \{f\}$;
13         end
14     end
15 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:

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

5 Coinciding Vertices

The `coincidingVertices` function is an essential component in mesh processing algorithms, designed to identify and handle vertices with the same coordinates. This functionality becomes crucial in scenarios where mesh data contains overlapping vertices due to precision limitations or data noise.

An example for the usage of `coincidingVertices` can be found in subsection 2.1. While we can see the four components of the teapot neatly displayed in Figure 4 this is after identifying similar vertices with each other. Just taking the original data and looking at the connected components results in a surprising amount of components.

```
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

print(f"Without␣coinciding␣the␣vertices␣there␣are␣{len(connectedComponents(teapot))
    }␣components.")

coincidingVertices(teapot)  # identifying and collapsing coinciding vertices
print(f"After␣coinciding␣the␣vertices␣there␣are␣{len(connectedComponents(teapot))}␣
    components.")
```

Code 1: Connected components with and without `coincidingVertices`

```
Without coinciding the vertices there are 19 components.
After coinciding the vertices there are 4 components.
```

What happened here is that the four intuitively expected components had been designed as quarters and then assembled. To the eye it does not make a difference but for the code it does.

Looking at some of the 19 components we see that the teapot seems to have been created by having parts of a repeating object, say the floor of the teapot where there are four quarters of, being moved and turned to fit together. For example, by running Code 7 without `coincidingVertices` in line 6 we end up getting said 19 components as .obj-files. Then running

```
from maniflow.mesh.parameterized import *
from maniflow.mesh.obj import OBJFile
from maniflow.render import SVGPainterRenderer
from maniflow.render.scene import *

# 4 times as we have four parts of the bottom of the teapot
for i in range(4):
    teapot_btm = OBJFile.read(f"teapot{16+i}.obj")
    # Invert orientation for colour visibility
    for j in range(teapot_btm.f):
        teapot_btm.faces[j].vertices = teapot_btm.faces[j].vertices[::-1]
    # Setting up renderer
    teapot_btm.setStyle('red')
    camera = Camera(position(10,-20,5), target=(0, -0.7, 0))
    scene = Scene(camer=camera, width=400, height=400, light=np.array([10, -1.5,
        50]
    renderer = SVGPainterRenderer(scene)
    # Take image and save it
    image = renderer.render(teapot_btm)
    image.save_svg(f"teapot_part{i+1}_render.svg")
```
Code 2: Visualizing the parts of the bottom of the teapot

gets us the four components that form the bottom of the teapot in Figure 6:

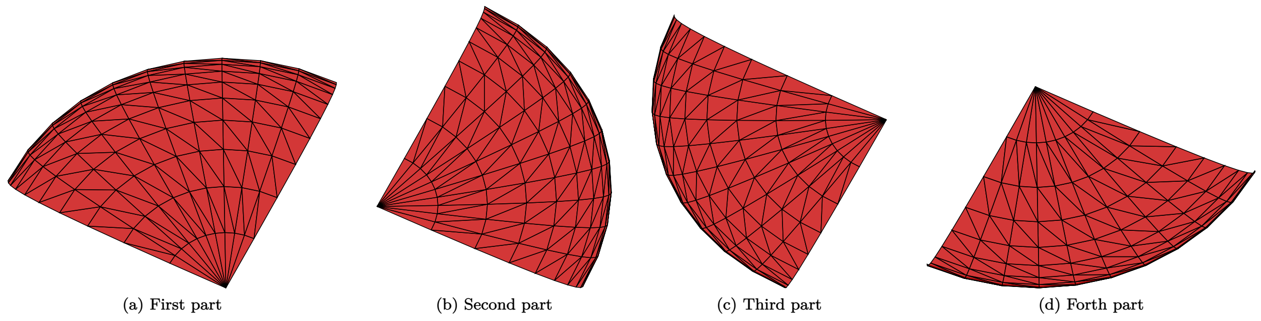

(a) First part (b) Second part (c) Third part (d) Forth part

Figure 6: The quarters of the bottom of the teapot.

Similarly, when we take a look at the moebius strip generated from a grid, we note that while the ends have the same points they still are not glued together. Again, to the eye there is no difference but topologically the moebius strip created from the grid is then just a plain with Euler Characteristic 1. Yet, it is well-known that an actual moebius strip has an Euler Characteristic of 0. Using **coincidingVertices** we can glue those loose ends together to get:

```
import numpy as np
from maniflow.mesh.parameterized import *

@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)

print(f"Euler␣Characteristic␣before␣coincidingVertices:␣{eulerCharacteristic(
    moebiusMesh)}")
coincidingVertices(moebiusMesh)
print(f"Euler␣Characteristic␣after␣coincidingVertices:␣{eulerCharacteristic(
    moebiusMesh)}")
```

Code 3: Euler Characteristic of the moebius strip with and without `coincidingVertices`

```
Euler Characteristic before coincidingVertices: 1
Euler Characteristic after coincidingVertices: 0
```

5.1 The Upper Triangular Approach

The old version of `coincidingVertices` utilizes an approach with a time complexity of $O(n^2)$, where n represents the number of vertices in the mesh. This stems from iterating twice through all or at least a linearly decreased subset of the vertex data. While being naive in its concept there are a few tweaks that improve the algorithm, more than halving its runtime. That of course is not as good as it sounds given that we have a quadratic runtime nature.

It starts by initializing empty data structures, iterates through each vertex to detect coinciding vertices, merges them into a single vertex, and updates the mesh accordingly. To reduce computational power needed this naive approach is improved by the simple fact that if $a == b$ then also $b == a$, so this halves the amount of iteration steps in a double for-loop leading to these computations. This leaves us with an upper triangular form where we only have to check the following indices but not the previous as those where checked before. E.g. when checking whether the third vertex can be identified with another vertex we can start at the fourth vertex since, say, the check against the second vertex already happened when we previously check for identities with the second vertex. The resulting checks can be visualized in a matrix looking similar to this

	1	2	3	4	5	n
1		X	X	✓	X	X
2			X	-	X	X
3				-	X	X
4					-	-
5						X
n						

where the 'X' implies a comparison to be done and the '✓' denotes, in this case, that the first vertex and the fourth vertex are the same point with regard to our tolerance. This results in canceling all checks for (row) and against (column) the fourth vertex as this check has been done for the identical first vertex already, indicated by '-'. Because of this canceling we get a best case scenario of $O(n)$ that is if all vertex are identified to be the same point. After one iteration through all of them we have identified them all with the first vertex resulting in not going through their rows and columns. This leaves only the first n checks.

	1	2	3	4	5	n
1		✓	✓	✓	✓	✓
2			-	-	-	-
3				-	-	-
4					-	-
5						-
n						

5.1.1 Bubblesort by accident

Without realizing at the time of first implementation the upper triangular approach is equivalent to the Bubblesort sorting algorithm. As a reminder, Bubblesort also first iterates through all elements. At the end of the iteration one of the elements is at its proper place. Afterwards, it repeats with the next iteration but this time it can leave out checking said element. Bubblesort functions in the following way:

Algorithm 4: Bubblesort

```
Input  : List of sortable elements
Output: Sorted List
Data: List
for i := 1 to n - 1 do
    Set swapped flag to False;
    for j :=1 to n - i do
        if j-th element is smaller than (j-1)-th element then
            Swap the two elements;
            Set swapped flag to True;
        end
    end
    if swapped flag is set to False then
        Stop;
    end
end
```

This is essentially the algorithm used for the upper triangular approach with the only difference being the direction of the inner loop, i.e. for the i-th row it starts from the at $i + 1$ and ends at the end n for the upper triangular but Bubblesort always starts at the beginning and ends at $n - i$.
The advantage we get now is that the runtime performance of Bubblesort is well known. Having the same structure this means that our upper triangular approach we ca not only confirm the performance mentioned above, that is $O(n)$ in the best case and $O(n^2)$ in the worst case, but also get the average case from Bubblesort, which is $O(n^2)$ as well.

5.2 Time to sort - Using Timsort to speed things up

The updated version of `coincidingVertices` employs a more efficient approach compared to the old version. It starts by creating an empty list `extVertList` and copies the mesh's faces into `faceList`. It then iterates through each face and its vertices, adding the j-th vertex in the i-th face along with the information on i and j to `extVertList`. Next, it sorts `extVertList` by the first element (the vertex coordinates) using Python's Timsort algorithm, resulting in a stable sorting with an average time complexity of $O(n \log n)$. Overall, timsort is linear with $O(n)$ in the best case as it is an adaptive sorting algorithm, and capped in the worst case at $O(n \log n)$.

The function then initializes an empty list `vertList` and iterates through all entries of `extVertList`. For each entry `v`, it checks if `vertList` is empty or if the vertex is not close to the previous one. If so, it adds the vertex to `vertList`. Additionally, it updates the vertices in `faceList` based on the indices in `vertList`, ensuring that vertices with the same coordinates are identified and glued together.

Algorithm 5: coincidingVertices Timsort

```
Input  : Mesh
Output: Mesh with identified vertices
Data: mesh
1  Initialize list 'extVertList';
2  Create work copy 'faceList' of the faces of the mesh;
3  for every face in 'faceList' do
4      for every vertex in the face do
5          Add [coordinate of vertex, index of face, index of vertex] to 'extVertList';
6      end
7  end
8  Sort 'extVertList' by coordinates, starting with the first;
9  Initialize list 'vertList';
10 for every entry 'v' in 'extVertList' do
11     if either 'vertList' is empty or the latest entry of 'vertList' is not close to 'v' then
12         Add 'v' to 'vertList';
13     end
14     Update the vertex in the face referenced in 'v';
15 end
16 Copy the values of 'faceList' into the initial mesh;
```

Note that the definition of closeness is not the standard metric but whether the vertices can be fitted into a box of size $2atol \times 2atol \times 2atol$. While not as intuitive as a simple distance between the points in the three dimensional, this allows us to use the a sorting algorithm since the vertices have to be within `atol` on *all* three coordinate axis. This means we can start looking at the first coordinate, compare that and, if it is within tolerance, we can continue with the second and third if needed. Hence, we can sort the first coordinates (if equal then go by the second, etc.). This approach can be used, as we did, to implement a fast algorithm that only needs to check the previous vertex for equality since it is the closest yet tested vertex, at least on the first coordinate. And since we are checking not the Euclidean distance between points but have this box approach all of the coordinates have to be contained. Thus, we can stop if the first coordinate is not close. If it is, we check for the second coordinate. As these are sorted as well, the latest entry to `vertList` has to be the closest to the currently checked vector in the second coordinate as well, given closeness in the first. This can be applied to the third coordinate as well.

By using not only the fast sorting of timsort but also being able to just check against a single different point (so comparing up to three different floats) this algorithm is very fast and efficient. Since we just iterate once through the faces and their three respective vertices ($3n$ operations), sorting with on average (and even in the worst case) $O(n \log n)$ and then iterate again only once though the list `extVertList` of length n, we end up with $O(3n) + O(n \log n) + O(n) = O(n \log n)$ as overall runtime.

5.3 Time comparison

A direct comparison can be drawn by using the `timeit`-package in Python. Since `coincidingVertices` normally overwrites the mesh given in the parameter we need to make adjustments to the functions for example introducing a variable called 'dummy' that is written into instead of the original mesh. It is only used for this writing of data and the timing method applied here.

The comparison will be drawn between the two implementations using the teapot OBJ-file already present. It consists of 3644 vertices which both functions will bring down to 3241. This is a decrease of about about 11.1% or one in every nine vertices being identified with another vertex.

As the main load of the computation is on the naive double-looping (upper triangle) and the sorting algorithm we can expect the other calls in the functions such as list manipulations and calls to stored data to be insignificant with the difference in performance being $O(n^2)$ and $O(n \log n)$ on average for a relatively (to simpler shapes) large $n = 3644$.

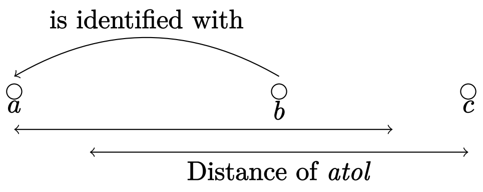
```python
from timeit import default_timer as timer

teapot = OBJFile.read("examples/teapot.obj")

repetitions = 10    # coincidingVertices_old is SIGNIFICANTLY slower than the newer
     version, choose repetitions accordingly
time_a = 0
time_b = 0

# cumulated time for the upper triangular method
for i in range(repetitions):
    start = timer()
    coincidingVertices_old(teapot)
    end = timer()
    time_a = time_a + (end - start)

# cumulated time for the timsort method
for i in range(repetitions):
    start = timer()
    coincidingVertices_new(teapot)
    end = timer()
    time_b = time_b + (end - start)

print(f"Upper␣Triangle␣Approach␣execution␣time␣avg.:␣{time_a/repetitions}␣seconds")
print(f"Sorting␣Algorithm␣Approach␣execution␣time␣avg.:␣{time_b/repetitions}␣
    seconds")
```

Code 4: Timing Comparison between Upper Triangular and Timesort Approach

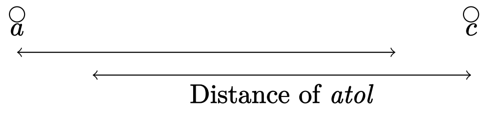
The code outputs the following result

```
Upper Triangle Approach execution time avg.: 81.20345090379979 seconds
Sorting Algorithm Approach execution time avg.: 0.33098463589976745 seconds
```

This result is completely within expectation and still depicts nicely how even for a not extremely large $n = 3644$ we already need about 245 times longer with the upper triangular than with timesort. Since we

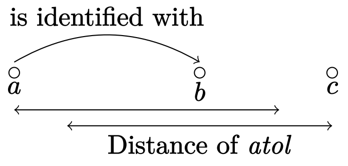


(a) Identifying b with a as we start with a

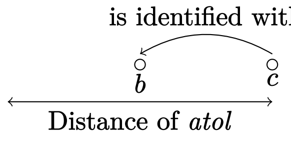


(b) a and c are not close enough for identification

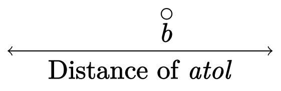
Figure 7: Coinciding the vertices a, b, c when starting with a



(a) Identifying a with b



(b) Identifying c with b



(c) Only b left, nothing to do

Figure 8: Coinciding vertices this time starting with b

are working with a non-small n the quadratic nature of the upper triangular approach weights heavy on the runtime to the point that using for the renderer the upper triangular approach is simply not feasible.

5.4 Further thoughts

With this speedy approach come limitations, though. The algorithm does not care for relative closeness, all it looks at is *whether* it is within `atol` but not *how* close the vertices are when within this tolerance. This can lead to cases where for three points $[a, b, c]$ with a and b being close (as in within the `atol`-box) and b and c being close, but not a and c.

The algorithm starts from a and checks against b. They are close so it identifies a and b with coordinates of a. But then a and c are not close so they are not the same (w.r.t. the program) even though b and c might have been closer to each other than a and b, see Figure 7. We hence can have non-optimal identifications for vertices which, again, is a trade-off that is made for increased computational efficiency.

Also note that the implemented function is iterating through the points in order that they are listed. Depending on the order in the list we can have different results visualized by comparing Figure 7 and Figure 8. So if we were to start from vertex b because it is mentioned before a and c, this means that we check whether $b == a$ and not $a == b$, then b and a would be identified in the point b and not a. As b and c are close as well, this would again be coincided in b resulting in one (b) and not two (a and c) remaining vertices originating from the initial three.

This could lead to the case that, as the upper triangle approach is simply iterating through the vertices in the mesh beginning from the first while the sort approach may change the order for the sorting, we may in some rarer cases end up with a different amount of vertices for the different implementations. The deviating numbers of vertices could become an issue if we were to use both approach alongside each other, which we are not. The upper triangle matrix was an early, naive approach to solve the problem for a small number of vertices where the $O(n^2)$ did not affect the computational time significantly. It will be completely replaced by the sort approach as the used timsort is better on average and in the worst-case with $O(n \log n)$ and equal in the best-cast (linear).

As another note, timsort is a stable sorting algorithm that means the even if we had points that are exactly the same the order in which they are mentioned would be kept so there are no differences if applied at different times despite of the shortcomings mentioned above.

6 Mesh simplification

Mesh simplification is a fundamental technique in computer graphics and computational geometry, employed to optimize the performance of various applications, such as 3D modeling, simulation, and rendering. By reducing the complexity of a mesh while preserving its essential features, mesh simplification achieves a balance between computational efficiency and visually staying true to the original.

The process of mesh simplification involves iteratively removing vertices, edges, or faces from the original mesh while minimizing the perceptible loss of detail. Various algorithms and methodologies have been developed to accomplish this task, ranging from straightforward decimation techniques to sophisticated error-driven approaches.

Furthermore, mesh simplification is not only beneficial for real-time rendering and interactive applications but also plays a crucial role in data storage, transmission, and manipulation. By generating simplified representations of complex meshes, the storage requirements are minimized, facilitating efficient data exchange and manipulation across different platforms and devices.

6.1 Implemented Approach

6.1.1 Approach to Mesh Simplification

In our approach we deploy vertex contraction as described in [2]. This way we cut down on the amount of vertices and hence faces. Technically, also the amount of edges is reduced but in our project the edges are only considered via vertices and faces.
The procedure has several steps depicted below as pseudocode:

Algorithm 6: Mesh Simplification

```
   Input  : mesh: A mesh
   Output: The simplified mesh
   Data: mesh
1  Clean the mesh to ensure consistency;
2  Initialize an empty list for Q values;
3  for each vertex in the mesh do
4      Compute the initial Q value for the vertex;
5      Add the computed Q value to the list;
6  end
7  Initialize an empty validity matrix for vertex pairs;
8  for each face in the mesh do
9      Mark vertex pairs sharing an edge as valid in the validity matrix;
10 end
11 Initialize an empty cost dictionary;
12 for each pair of valid vertices do
13     if the normals of the adjacent faces don't flip after contraction then
14         Calculate the cost of contraction and add it to the dictionary;
15     end
16 end
17 Sort the cost dictionary based on cost values;
18 Initialize a reduction goal based on desired reduction percentage;
19 while the number of faces in the mesh is greater than the reduction goal do
20     Choose the vertex pair with the lowest cost from the dictionary;
21     Update the mesh by contracting the chosen vertex pair;
22     Update the validity matrix and cost dictionary accordingly;
23 end
24 Remove redundant vertices and faces from the mesh;
25 return the simplified mesh;
```

After getting rid of redundant vertices by applying `coincidingVertices` we want to compute a characteristic matrix $\boldsymbol{Q}$ for each vertex. This characteristic will help us in computing the optimal contraction point. Say, we have a vector $\boldsymbol{v} = [v_x, v_y, v_z, 1]^T$. Then the error at $\boldsymbol{v}$ is denoted by

$$\begin{aligned}\Delta(\boldsymbol{v}) &= \sum_{\boldsymbol{p} \text{ is a plane of } \boldsymbol{v}} (\boldsymbol{v}^T\boldsymbol{p})(\boldsymbol{p}^T\boldsymbol{v}) \\ &= \boldsymbol{v}^T \left(\sum_{\boldsymbol{p} \text{ is a plane of } \boldsymbol{v}} \boldsymbol{K_p} \right) \boldsymbol{v}\end{aligned}$$

with $\boldsymbol{K_p} = \boldsymbol{pp}^T$ being called the fundamental quadric error. Since $\boldsymbol{p}$ is a plane given by the equation

$$ax + by + cz + d = 0,\ a^2 + b^2 + c^2 + d^2 = 1$$

we get that $\boldsymbol{K_p}$ is a 4×4-matrix.
The characteristic matrix $\boldsymbol{Q}$ mentioned before is then the sum over fundamental quadric errors. We get that

$$\Delta(\boldsymbol{v}) = \boldsymbol{v}^T\boldsymbol{Q}\boldsymbol{v} \tag{3}$$

and choose that for two vertices $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$ and their respective $\boldsymbol{Q_1}$ and $\boldsymbol{Q_2}$ we have $\bar{\boldsymbol{Q}} = \boldsymbol{Q_1} + \boldsymbol{Q_2}$ in their contraction point $\bar{\boldsymbol{v}}$. Minimizing the error $\Delta(\bar{\boldsymbol{v}})$, we use that it is quadratic. That implies that the minimal solution to this problem is linear, meaning that we set the partial derivatives of the error to zero. The problem can be reformulated to

$$\bar{\boldsymbol{v}} = \begin{pmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{21} & q_{22} & q_{23} & q_{24} \\ q_{31} & q_{32} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

where q_ij is the element of the i-th row and j-th column of $\bar{\boldsymbol{Q}}$.

If the matrix is not invertible then we want to choose one of the two initial vertices or their midpoint as new point of contraction. This depends on whichever of the three is minimizing the error best among them.

For the next step we want to get *valid pairs* of vertices. Two vertices form a *valid pair* if either they are belonging to the face, i.e. they are connected by an edge, or, if they are closer to each other than a given parameter `tol`. Only these *valid pairs* are considered for contraction.

But an object has many edges. How do we determine which of those *valid pairs* is the best to go for or start with? This is done by coming up with a cost function for the contraction of a *valid pair.* We want to then work with the pair with the lowest cost of contraction. This is where we can apply Equation 3.

Definition 5. *We define our cost function to be the error at the contraction of the potential valid pair* $[\boldsymbol{v_1}, \boldsymbol{v_2}]$ *with their respective characterizing matrices* $\boldsymbol{Q_1}, \boldsymbol{Q_2}$, *i.e.*

$$\Delta(\bar{\boldsymbol{v}}) = \bar{\boldsymbol{v}}^T(\bar{\boldsymbol{Q}})\bar{\boldsymbol{v}} = \bar{\boldsymbol{v}}^T(\boldsymbol{Q_1} + \boldsymbol{Q_2})\bar{\boldsymbol{v}}$$

Now we iterate through all the valid pairs, compute their cost and then sort them from smallest to largest value of the cost function. Our next contraction is simply the pair which has the lowest cost. One thing we have to consider though is whether an inversion of parts of the mesh happens if this pair contracts. This would be against the idea of mesh simplification as we would not represent the original mesh faithfully. So we take the pair with the lowest cost, do the contraction and check for inversions via the normals of the faces, i.e. the `normal` property of the `face`-class. If an inversion occurs we reverse this step, ignore this pair and move on to the next and so on.

After having contracted a pair of vertices, we need to update the adjacent vertices and faces to this new point. Say the vertices involved in the contraction were v_1 and v_2. Then we go through all of their adjacent faces and see whether they share a face. If that is the case then an edge was contracted from a face meaning that the face itself is to be deleted. If a face has only one of them as vertex then we can just update the coordinates of this vertex to the contraction point calculated earlier. Finally, we recalculate the cost of valid pairs that included either v_1 or v_2, i.e. share and edge with the new contraction point. We repeat this until the number of faces is sufficiently low, as instructed by the parameter given by the user.

6.1.2 Overview of the functions in the code

Our implemented code aims to perform mesh simplification by cleverly reducing the complexity of input meshes while preserving their essential geometric characteristics.
The implementation consists of several smaller functions that each do their part for the larger picture. The main function itself is `simplifyByContraction`. This overview aims to provide a rough idea on the smaller parts of the code.

1. **Computing Plane Equation Coefficients**
   The `computePlaneEquation` function computes the coefficients of the plane equation given three vertices. It utilizes vector calculations and the cross product to determine the plane's normal vector, which is then normalized to ensure unit length. The coefficients are returned in a list conforming to the equation $ax + by + cz + d = 0$, where $a^2 + b^2 + c^2 + d^2 = 1$.

2. **Computing Fundamental Error Quadric**
   The `computeFundamentalErrorQuadric` function calculates the fundamental error quadric $\boldsymbol{K_p} = \boldsymbol{pp}^T$ for a plane represented by its coefficients. This quadric matrix is essential for characterizing the error associated with a plane. Also the sum over those $\boldsymbol{K_p}$ is denoted by the characteristic matrix $\boldsymbol{Q}$.

3. **Computing Initial Characterization $\boldsymbol{Q}$ of Vertex Error**
   The `computeInitialQ` function computes the initial characterization of error at a vertex. It sums the error quadrics of all adjacent faces to the vertex, providing a comprehensive assessment of the vertex's error.

4. **Optimal Contraction Point Calculation**
   The `optimalContractionPoint` function determines the optimal point for contracting two vertices. It takes into account the error matrices associated with the vertices and calculates the contraction point accordingly. In cases where the error matrices are not invertible, it computes a midpoint between the vertices as an alternative contraction point.

5. **Vertex Contraction and Mesh Rewriting**
   The `rewriteFaces` function performs vertex contraction and updates the mesh accordingly. It adjusts adjacent faces' vertices and recalculates the position of the contracted vertex based on the optimal contraction point.

6. **Checking Normals Consistency**
   The `normalsFlipped` function verifies the consistency of normals before and after vertex contraction. It ensures that the contraction does not result in flipped normals, which could distort the mesh's geometry.

7. **Cost Evaluation for Vertex Contraction**
   The `contractingCost` function evaluates the cost associated with contracting two vertices. It utilizes the error matrices and contraction point to compute the contraction cost, aiding in identifying low-cost contractions for mesh simplification.

8. **Generating Valid Vertex Pairs**
   The `getValidPairs` function generates a validity matrix indicating valid pairs of vertices in the mesh. Validity is determined by vertices sharing an edge or being sufficiently close together, facilitating the contraction process.

9. **Mesh Simplification by Contraction**
   The `simplifyByContraction` function orchestrates the mesh simplification process using vertex contraction. It iteratively identifies and contracts vertex pairs based on cost considerations, aiming to achieve the desired reduction in mesh complexity while preserving its essential features.

6.2 Outlook and Future Works

When first looking into mesh simplification other approaches and methods can be found. A first attempt was made in reference to "Mesh optimization"([3]). While the paper focuses on edges and `maniflow` does only implicitly work with edges via faces and vertices one can still transfer the operations from one system to another. It introduces three edge operations to optimize the mesh,

- Edge collapse (equivalent to vertex contraction)
- Edge split (introducing a new vertex on the edge)
- Edge swap

The goal of the paper is to get a mesh that characterizes given data well enough or better while at the same time being simpler than the initial mesh.

Sadly, the approach is not easily implemented for the scope of `maniflow`. As mentioned before, since it is mesh optimization and not simplification, even though one might lead to another, we have an initial mesh and data points that we want to be represented as good as possible by the mesh. In `maniflow` our mesh does already represent the data, that is how the mesh is formed in the first place.

The code can be written and tested to work with all kinds of faces independent of the amount of vertices they have. E.g. the current three vertices implementation results in a degenerate face if we contract two vertices of a face. But for a face consisting of four vertices, the shape of the face would just change from a square to a triangle having now three vertices. It is easy to change the number of iterations needed in some for-loops but more complicated when being confronted with several iterations of contractions on a single face as it loses one vertex each time.

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`.

[2] Michael Garland and Paul S. Heckbert. “Surface Simplification Using Quadric Error Metrics”. In: *Seminal Graphics Papers: Pushing the Boundaries, Volume 2*. 1st ed. New York, NY, USA: Association for Computing Machinery, 2023. ISBN: 9798400708978. URL: `https://doi.org/10.1145/3596711.3596727`.

[3] Hugues Hoppe et al. “Mesh optimization”. In: *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’93. Anaheim, CA: Association for Computing Machinery, 1993, pp. 19–26. ISBN: 0897916018. DOI: `10.1145/166117.166119`. URL: `https://doi.org/10.1145/166117.166119`.