POLITECNICO DI TORINO

Master’s degree in computer engineering

Thesis

Study of remeshing for 3D human models using an AI-based approach

Supervised by:

Prof. Andrea Sanna
Prof. Federico Manuri

by:

Fabio Alfredo Chavez Energici

a.a. 2024/2025

ABSTRACT

When creating 3D models for real time applications a crucial step in their development is the optimization of the mesh, reducing the number of needed vertices to represent it and in some cases apply a specific topology to it. When dealing with specific kinds of models some special characteristics maybe needed to be applied to said mesh. This task normally involves experts who use multiple hours improving meshes to end with a good final product. Even if some automation tools exist, they are not the norm and normally still require user input to provide the best results. In recent years the use of geometric neural networks has become a topic of interest, and the use of this technology may prove useful automating some of the steps needed to provide good meshes automatically. We focused this thesis in investigating what types of remeshing exist, how they can be applied in such a way that they create a surface with a specific topology and how could we parametrize them in such a way that a machine learning algorithm could tackle them. We divided the whole process into a study on how to remesh, a study on the state-of-the-art on geometrical neural networks and proposed an architecture to provide a frame field that can be used to automate remeshing for human models. The results unfortunately were not precise enough to provide a good final product, but the approach seems promising, and we estimated that given enough training time we could reach results that give a mesh with the desired topology.

CONTENTS TABLE

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

DDG Discrete Differential Geometry

LOD Level of Detail

RoSy Rotationally Symmetric

DOF Degrees of Freedom

NN Neural Network

MLP Multi-Layer Perceptron

AI Artificial Intelligence

FAUST Fine Alignment Using Scan Texture

DFAUST Dynamic FAUST

INTRODUCTION

This thesis was suggested as a first step to improve remeshing algorithms and taking advantage of developing technologies, specifically neural networks. This thesis focuses mostly on collecting the relevant data to understand geometric computing and researching state-of-the-art algorithms which can utilize neural network in their pipeline to create meshes with a given topology. Human meshes are a relevant topic as their wide use in different fields, from animation to real time applications and even simulations, makes them a prime subject for a standardized mesh, with a generally agreed on topology and industry use, their automatization would be beneficial on a wide scale.

We divided this thesis into parts to allow for future research to simplify the study of remeshing and how it can relate to neural networks.

The first part is a study of discrete differential geometry, a complex topic which is necessary to understand most algorithms about remeshing and working with 3D meshes in general.

The second part is a chapter summarizing the most common techniques used for remeshing in general and their implications. We reviewed the general remeshing algorithms applied to triangular meshes and separately we review algorithms dealing with quadrangular meshes. Here we also added a small chapter exploring frame fields and directional fields, their characteristics and challenges. Frame fields have proven to be a good approach when trying to create quadrangulations of existing meshes, their application to create a specific topology was the main reason for their study.

After this, a review of neural networks in general was done, after which led to the investigation of geometric neural networks, their implementation, classification and projection of a possible model to test the research done.

Finally, we proposed a model to predict frame fields which could lead to automatization of remeshing algorithms and simplification of the full modelling pipeline. The proposed architecture was evaluated on its precision predicting frame fields on a triangular mesh and its correspondence with a ground truth. This part of the study also provides the adaptation of an existing database such that future research on the topic may approach it without the need to prepare it from scratch.

1 INTRODUCTION TO DISCRETE DIFFERENTIAL GEOMETRY

Due to the nature of the data we wish to analyze, an introduction to discrete differential geometry (DDG) seemed appropriate. Specifically, we will define what a mesh is, give a brief introduction to what properties we wish for a good mesh and some of the tools that are utilized in their analysis. We will explore the fundamental definitions of discrete differential geometry the tools used on its analysis and provide a basis to understand general algorithms or tools that may be used in them. Furthermore, later we will introduce the concept of directional fields. These fields work on top of surfaces and require some decisions on their construction based on DDG properties, that we will briefly introduce here. For an in-depth presentation, we recommend [1].

1.1 MESH DEFINITIONS - TERMINOLOGY AND DATA STRUCTURES

A Mesh is a representation of a 3D object in space. Meshes can be subdivided into volume meshes and surface meshes. We will only deal with polygonal surface meshes and we will refer to them simply as "mesh" or "meshes". A mesh is formed by **vertices, edges,** and **faces** or facets. Later, we will introduce the concept of simplices but the important thing to keep in mind about how they relate to meshes is that a vertex will be a 0-simplex, an edge a 1-simplex and a face a 2-simplex. Furthermore, meshes are considered the union of geometry and topology. The geometry defines the position of the vertices in space and the shape that its surface forms, so two meshes have the same geometry if they have the same surface, but not necessarily the same set of vertices and edges. The topology provides all the information referring to the connections between different adjacent vertices.

A vertex is defined as a point in the 3D space, with coordinates defined in an orthonormal basis X, Y, Z.

An edge is an element formed by the union of two vertices and has a direction going from the first vertex V_A towards the second vertex V_B. These edges will be written as an ordered tuple (V_A, V_B) or (V_B, V_A), depending on the orientation.

In some cases, algorithms work with half-edges, which are defined as the signed part of an edge belonging to a face. This definition brings the possibility that a single edge may have two half-edges with opposite directions; for the purposes of this thesis, we will not make major difference between these two terms.

A face or facet is defined as a polygon formed by a cycle of edges, which in turn means it can be written as a chain of vertices. Faces also have direction, given that to define the face vertices are written in order. We can follow the cyclic order of the definition to define the normal direction of each face. The reason why half-edges must exist is that, otherwise, two contiguous faces would only be allowed to have the same relative orientation. For our purposes we purposefully made sure all working data used by us had this property, so if at any point we mention the need to work on half-edges is only to invert the orientation of one of the adjacent faces to a specific edge.

We primarily work with *conforming* meshes in which two faces may only share either a single vertex or at most a single edge, and edges can belong to a maximum of two faces.

In total we can refer to a mesh as:

$$\mathcal{M} = \{(\mathcal{V}, \mathcal{F}) | \mathcal{V} = \{v_i\}_{i=1,2,\dots,V}, \mathcal{F} = \{f_i\}_{1,2,\dots,F}, f_i \in \mathcal{F}^d\}$$

Such that $\mathcal{M}$ is a mesh, $\mathcal{V}$ is the set of V vertices that compose it, $\mathcal{F}$ is the set of faces, and d denotes the number of vertices corresponding to each face.

A T-mesh is defined as a mesh which has T-joints; for our purposes the T-mesh does not have to be aligned with regular meshes in the sense that is formed by the same vertices, edges or facets, but it will follow the same geometric surface. T-meshes for our purposes are to be thought of as a superposition on the original mesh and are the union of square patches on our surface. They have the same genus as the original input mesh, and their structure can be described by a cyclic graph where each node corresponds to a T-joint, and each edge is the geodesic line which unites two T-nodes. For a more detailed definition, we recommend [2].

The *valence* of a vertex is defined as the number of incident edges on it, while the *star* of the same vertex is the set conformed by both faces and edges which connect to it. This allows us to define our meshes to be manifold if, for every star belonging to our mesh, the star can be defined to be homomorphic to either a disc or a half plane. In other words, every vertex in our mesh can belong to one and only one star and any edge belonging to our mesh can belong to either one or at most two faces: if they have only one incident face, that edge is defined as a *boundary,* otherwise we call it *internal.* If a mesh is conformed exclusively by internal edges its defined to be "*watertight*".

Meshes can be categorized by the basic shapes that construct them. A mesh exclusively formed by triangles is defined as a *Trimesh* or *triangle mesh,* while one formed only by quadrilaterals (also called ***quads***) is called a quad mesh or quad-mesh. A quad mesh can be defined as a quad-dominant mesh if there are other polygons besides quads as faces. We will refer to the number of polygons that form a mesh as polycount. Any polygon different from a quad or a triangle generally will be referred to as an "***N-gon****"*.

Is important to clarify that any N-gon can be converted to, at worse, a set of N-triangles, by creating a vertex in its center an "cutting" it from each vertex towards the center. This is useful, since for some algorithms it is necessary to work with trimeshes. When creating new facets from preexisting ones it is necessary to conserve the normal orientation, this can be done easily if any edge of the previous face is conserved, otherwise it becomes a duty of the algorithm to either provide a way to conserve orientation or specify that it doesn't ensure it.

A tangent bundle is defined as the collection of all the tangent spaces for all points of a manifold. Informally, in the case of our meshes, we defined our "*discrete tangent bundle"* as the set of orthonormal bases defined using the normal of each face belonging to our mesh, meaning we have a discontinuous tangent bundle since is not well defined on nether vertices nor edges. But it can be expanded to fit them with some conditions.

1.2 POINT CLOUDS

A point cloud is defined as a set of data points in a 3D coordinate system. Each point represents a precise location in space. These points may represent a surface in 3D space, but their connectivity is unknown. Each point may contain feature data besides its coordinates, like an RGB color, normals, timestamps or any given value. They are highly available as they can be produced by 3D scanners. They can be converted into meshes, and it is common to do so, but the quality of these meshes tends to be low without human intervention. It is common for point clouds to have more than one object per file, and the distinction of these objects tends to be a challenge, and when creating meshes their separation tends to be rather complicated. They are a common dataset used to train neural networks.

Points clouds have three main properties:

Point clouds are an unordered set of points, meaning that any process which deals with point clouds needs to be invariant to permutations.

The points exist in a space with a given distance metric. Meaning points are not isolated and neighboring points create local structures. This can be implicit information as generally when studied in neural network training is a task of the network to capture these local structures.

Point clouds should be invariant under regular transforms. Meaning if the whole set of points forming a point cloud suffers a transformation (mainly rotations, translations and scaling) the total information about the set should remain the same. For neural networks this means the predictions should remain invariant to any transformation.

1.3 DUAL SPACE OF A MESH

When working with meshes we have the called mesh dual space, which is a space transforming the components that form a mesh into equivalent "*dual*" dimensioned values that corresponds to them. Our original mesh will be called a primal mesh, and its conforming values (faces, vertices, and edges) will be preceded by the word "*primal*". The *dual* mesh or *Poincaré dual* will be formed similarly by having dual faces (or cells), edges and vertices. Specifically, the correspondence goes as:

For every primal vertex in our primal mesh, we will have a corresponding cell. This cell is formed by connecting adjacent triangles to the initial vertex in the primal mesh. In our case specifically, we will connect their barycenters, forming a polygon centered around the initial vertex. This way of creating a dual cell is arbitrary, a second attractive option was to use the circumcenter as the vertices of the new cell. But since it created some edge cases on obtuse triangle it was decided to avoid it.

For every primal edge in the primal mesh, we will have a corresponding dual edge. These dual edges will be formed by the connection between the two adjacent triangles to the original primal edge. When working with watertight meshes this is always possible. In case we have borders, on

our mesh, a common solution is to use a point in the border triangle and extend a line from it through a point on the primal edge. For our specific case, we know we work with watertight meshes, we will connect the barycenters of the two primal triangles adjacent to each edge.

Finally, for every triangular primal face, in our primal mesh, a corresponding dual vertex exists. This dual vertex, in our specific case, is in the barycenter of the primal face.

Using these definitions, we have a corresponding dual mesh, where its dual vertices form its dual edges, and its dual edges form its dual cells. It is important to note that this property is not necessarily true for all dual meshes, but only for those where the points defining its components are the same.

1.4 CALCULUS ON MESHES - EXTERIOR CALCULUS

Exterior calculus is a field of math that allows us to extend integration and differentiation to multidimensional manifolds. It is useful as it can be easily translated to a discrete setting allowing for use of calculus on discrete surfaces.

1.4.1 Exterior algebra

Before explaining explicitly how to apply calculus on meshes it is necessary to define some basis for it. First, we need to define concretely exterior algebra (the formal definition will be given at the end of this section). And to do so it would be better to provide a general understanding of the concepts of k-vectors, the wedge product and the hodge star operator.

1.4.1.1 Wedge product (∧) and k-vectors

Given two vectors u and v, the wedge product (or exterior product) between them ($u \wedge v$) describes the oriented area of a parallelogram with sides u and v. The wedge product of k vectors $v_1 \wedge v_2 \wedge \ldots \wedge v_k$ is called a k-blade or k-vector. The magnitude of a k-blade is the (hyper)volume of a parallelotope defined by the previously mentioned vectors.

Very importantly this allows us to define signed areas as 2-vectors and signed volumes as 3-vectors.

It has skew symmetry, meaning:

$$u \wedge v = -v \wedge u$$

And $v \wedge v = 0$, generally.

The wedge product is associative and distributive:

$$u \wedge v \wedge w = (u \wedge v) \wedge w = u \wedge (v \wedge w)$$

$$v_1 \wedge u + v_2 \wedge u = (v_1 + v_2) \wedge u$$

Importantly k-vector are defined exclusively by a direction and a magnitude, with the exception of "0-vectors" which are scalar values.

1.4.1.2 Hodge star product and the Hodge star $(\star)$

The hodge star product or simply Hodge product gives us the oriented orthogonal complement of a linear subspace.

Let $U \subseteq V$ be a linear subspace of a vector space V with inner product $\langle \cdot, \cdot \rangle$. The orthogonal complement of U is the collection of vectors $U^{\perp} \coloneqq \{v \in V \mid \langle u, v \rangle = 0 \, , \forall \, u \in U\}$

This operator allows us to find an equivalence between wedge product and the normal vector describing the signed area of the wedge product, as seen in Figure 1. The Hodge star product is represented by the Hodge star or hodge star operator $(\star)$.

$$\star \, (u \wedge v) = w$$

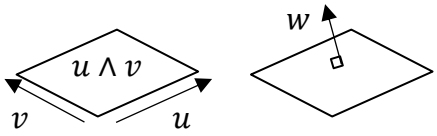


Figure 1 visual representation of the hodge star

By convention $z \wedge \star \, z$ is positively oriented. When working in 2D, by convention the hodge product follows the right-hand rule to define positive orientation. Meaning that the hodge star of a single vector in 2D is a counterclockwise 90-degree rotation of said vector.

1.4.1.3 Basis using k-vectors

A k-vector basis can describe any k-vector within it as a linear combination of the k-vectors that form that basis or a combination using the wedge product of n-vectors where n<k.

More concretely formal definitions of the previously presented terms are:

Let $e_1, \ldots, e_n$ be the basis for an n-dimensional inner product space V. For each integer $0 \leq k \leq n$, let $\wedge^k$ denote an $\binom{n}{k}$-dimensional vector space with basis elements denoted by $e_1 \wedge \ldots \wedge e_{i_k}$ for all possible sequences of indices $1 \leq i_1 < \cdots < i_k \leq n$, corresponding to all possible "axis-aligned" k-dimensional volumes. Elements of $\wedge^k$ are called k-vectors.

The *wedge product* is a bilinear map

$$\wedge_{k,l} \colon \wedge^k \times \wedge^l \to \wedge^{k+l}$$

Uniquely determined by its action on basis elements; in particular, for any collection of *distinct* indices $i_1, \ldots, i_{k+l}$,

$$\left(e_{i_1} \wedge \ldots \wedge e_{i_k}\right) \wedge_{k,l} \left(e_{i_{k+l}} \wedge \ldots \wedge e_{i_{k+l}}\right) \coloneqq sgn(\sigma) e_{\sigma(i_1)} \wedge \ldots \wedge e_{\sigma(i_{k+l})}$$

Where σ is a permutation that puts the indices that puts the indices of the two arguments in canonical order. Arguments with repeated indices are mapped to $0 \in \wedge^{k+l}$. For brevity, normally the subscript on $\wedge_{k,l}$ is dropped.

Finally, the *Hodge star on k-vectors* is a linear isomorphism

$$\star\colon \wedge^k \to \wedge^{n-k}$$

Uniquely determined by the relationship

$$\det(\alpha \wedge \star\, \alpha) = 1$$

Where α is any k-vector in the corresponding k-basis, $\alpha = e_{i_1} \wedge \ldots \wedge e_{i_k}$ and denotes the determinant of the constituent 1-vectors (column vectors) with respect to the inner product on V. The collection of vector spaces $\wedge^k$ together with the maps $\wedge$ and $\star$ define an *exterior algebra* on V, sometimes known as *graded algebra.*

1.4.2 *Dual* space

Let V be any real vector space. Its *dual vector space* V^* is the collection of all linear functions $\alpha\colon V \to \mathbb{R}$ together with the operations of *addition* and *scalar multiplication.* Meaning:

$$(\alpha + \beta)(u) \coloneqq \alpha(u) + \beta(u)$$

$$(c\alpha)(u) \coloneqq c(\alpha(u))$$

Where $\alpha, \beta \in V^*, u \in V$, and $c \in \mathbb{R}$. All elements in a dual vector space are called *dual vectors* or *covectors*.

1.4.2.1 Sharp (♯) and flat (♭) operators

The sharp and flat operators allow for the mapping from vector space to its dual space. They can be thought of as similar to the transpose of a vector for Euclidean spaces.

Given $u, v \in V$ then applying the ♭ operator we can convert one of them into a covector.

$$u, v \to u^{\flat}(v)$$

Similarly, given $\alpha, \beta \in V^*$ we can transform them into corresponding vectors using the ♯ operator.

$$\alpha, \beta \to \alpha(\beta^{\sharp})$$

These operations are particularly useful when the inner product is defined via a *Mass matrix (M)*. We can say that:

$$\langle u, v \rangle = u^{\mathrm{T}} M v = u^{\flat}(v)$$

Similarly, we can define the complementary dual operation as

$$\alpha(\beta^{\#}) = \alpha M^{-1}\beta^{\mathrm{T}}$$

M is particularly important when working with curved geometry as it defines the inner product on non-Euclidean surfaces.

1.4.3 *k*-Forms

k-forms are the dual of k-vectors. k-forms are analogous to covectors in their use, as they can be seen as a fully antisymmetric multilinear measurement of a k-vectors. Geometrically measuring using a k-form, is like calculating the size of the projections of a k-vector into a k-form. They can be used as a map $\alpha: V_1 \times \ldots V_k \to \mathbb{R}$. The 0-forms are scalars.

e.g., given the 2-form formed by the wedge product of the 1-forms (covectors) α, β, we can measure the area of the projection of the 2-vector formed by the wedge product of u and v as:

$$u_{\alpha\beta} = (\alpha(u), \beta(u))$$

$$v_{\alpha\beta} = (\alpha(v), \beta(v))$$

Where $u_{\alpha\beta}$ and $v_{\alpha\beta}$ are the projections of u and v respectively on the plane defined by α and β. Then the measured area can be calculated by the cross product of $u_{\alpha\beta}$ and $v_{\alpha\beta}$. Meaning that

$$(\alpha \wedge \beta)(u, v) \coloneqq \alpha(u)\beta(v) - \alpha(v)\beta(u)$$

This example can be generalized to k dimensions,

$$(\alpha_1 \wedge \ldots \wedge \alpha_k)(u_1, \ldots, u_k) \coloneqq \det\left(\begin{bmatrix} \alpha_1(u_1) & \cdots & \alpha_1(u_k) \\ \vdots & \ddots & \vdots \\ \alpha_k(u_1) & \cdots & \alpha_k(u_k) \end{bmatrix}\right)$$

When working in an n-dimensional vector space V. We can express vectors v in a basis $e_1, \ldots, e_n$ as

$$v = \sum_{i=1}^{n} v^i e_i$$

The scalar values v^i correspond to the coordinates of v.

Similarly, we can write covectors α in a *dual basis* $e^1, \ldots, e^n$ as

$$\alpha = \sum_{i=1}^{n} \alpha_i e^i$$

The relationship between the canonical basis and the corresponding dual basis is described by

$$e^i(e_j) = \begin{cases} 1, & i = j \\ 0, & otherwise \end{cases}$$

Given this relationship, we can use Einstein summation notation, describing that whenever we have an evaluation of a vector using a covector, all terms which $i \neq j$ are zero, therefore we can summarize it as

$$x^i y_i \coloneqq \sum_{i=1}^{n} x^i y_i$$

Conveniently this aligns neatly with the musical isomorphisms #, ♭. where # raises the index and ♭ lowers it (similarly as how they work in music).

1.4.4 Differential k-Forms

Differential k-forms or *k-simplices* can be described as the assignation of a k-form to every point in a space.

A differential 0-form is a scalar function. A differential 1-form can be visualized similarly as a vector field but is more accurate to describe it as "how to measure a vector field at each point", it generally is described as how strong is the flow of the vector field X, along the direction α. In general differential k-forms apply the corresponding k-form to each point.

We can describe the differential k-form α in a dual basis dx^k, where

$$\alpha = \sum_{i=1}^{k} \alpha_i dx^i$$

Vector fields can be written, similarly, in a basis $\frac{\partial}{\partial x^k}$, where if we have a vector field X it can be written as

$$X = \sum_{i=1}^{k} x_i \quad \frac{\partial}{\partial x^i}$$

The relationship between these two bases is

$$dx^i\left(\frac{\partial}{\partial x^j}\right) = \delta_j^i \coloneqq \begin{cases} 1, & i = j \\ 0, & otherwise \end{cases}$$

(**Important:** in this case the notation is just based on convention and does not denote in any way, shape or form the idea of a derivative or differentiation)

In n-dimensions, any *positive* multiple of $dx^1 \wedge \ldots \wedge dx^n$ is known as a *volume form.*

1.4.5 Exterior Derivative

Let us call Ω^k the space of all differential k-forms, then the exterior derivative can be defined as a unique *linear* map $d : \Omega^k \to \Omega^{k+1}$, such that for

$$k = 0, \qquad d\phi(X) = D_X\phi$$

Meaning that the exterior derivative of ϕ returns a 1-form which when applied to a vector field X gives the same result as taking the directional derivative of ϕ along X.

The exterior derivative must also comply with the product rule and when applied to the wedge product of two k-forms it will be

$$d(\alpha \wedge \beta) = d\alpha \wedge \beta + (-1)^k \alpha \wedge d\beta$$

And finally, the exterior derivative must have **the exactness property**, defined as

$$d \circ d = 0$$

Coordinate wise the exterior derivative of ϕ can be defined as the sum of its partial derivatives in the canonical basis directions.

$$d\phi = \sum_{i=1}^{n} \frac{\partial \phi}{\partial x^i} dx^i$$

Importantly the gradient, in exterior calculus, is defined as

$$\langle \nabla_\phi, X \rangle = D_X\phi$$

Meaning that: the gradient is the unique vector field ∇_ϕ whose inner product with any vector field X, returns the directional derivative $D_X\phi$ along X.

Importantly, although similar, the gradient is a vector field, while the exterior derivative is a differential 1-form. And the gradient is defined and is dependent on the inner product while the differential doesn't. Their relationship can be written as

$$d_\phi(\cdot) = \langle \nabla_\phi, \cdot \rangle$$

Meaning that by definition

$$(d\phi)^\# = \nabla_\phi$$

$$\left(\nabla_\phi\right)^\flat = d_\phi$$

There are some quantities that we wish to mention as particularly convenient to write using differential k-forms: these are the gradient, the curl and the divergence.

Using the 1-form $\alpha := udx + vdy + vdz$, where u, v, w are each scalar a function $\mathbb{R}^3 \to \mathbb{R}$. The differential $d\alpha = \left(\frac{\partial w}{\partial y} - \frac{\partial v}{\partial z}\right) dy \wedge dz + \left(\frac{\partial u}{\partial z} - \frac{\partial w}{\partial x}\right) dz \wedge dx + \left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right) dx \wedge dy$

Similarly, given the vector field $X := u\frac{\partial}{\partial x} + v\frac{\partial}{\partial y} + w\frac{\partial}{\partial z}$, its curl can be expressed as

$$\nabla \times X = \left(\frac{\partial w}{\partial y} - \frac{\partial v}{\partial z}\right)\frac{\partial}{\partial x} + \left(\frac{\partial u}{\partial z} - \frac{\partial w}{\partial x}\right)\frac{\partial}{\partial y} + \left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right)\frac{\partial}{\partial z}$$

With this similarity becomes convenient to write that:

$$\nabla \times X = \left(\star\, dX^{\flat}\right)^{\#}$$

Meaning that if we use the $\flat$ operator to transform X into its corresponding 1-form, we apply the exterior derivative obtaining the 2-form shown above, we apply the hodge star operator obtaining the complementary 1-form, and finally applying the # operator we can get the exact same result.

Similarly, always with the same α the differential of $\star\,\alpha$ is

$$d \star \alpha = \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right) dx \wedge dy \wedge dz$$

And noted that the divergence of X is

$$\nabla \cdot X = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}$$

If we define $\alpha = X^{\flat}$, then simply by applying the hodge star operator we can say that

$$\nabla \cdot X = \star\, d \star X^{\flat}$$

To abbreviate notation, it's been called a *codifferential* δ *to the operation* $\star\, d \star$.

Finally, the gradient can directly be defined using a scalar function ϕ, its gradient can be written as

$$grad\ \phi = (d\phi)^{\#}$$

1.4.6 Integration of differential forms

Since differential forms naturally deal with areas and volumes, integration becomes a natural step for them and does not require mayor new definitions with respect to classical integration. The one point we will mention is that for integration the wedge product can act more naturally with classical notation when specifying 2-forms. E.g. if we integrate $\omega := (x + xy)dx \wedge dy$ in a domain Ω, it can be expressed as double integral along dx and dy.

$$\int_{\Omega} \omega = \int_{\Omega} (x + xy)dx \wedge dy = \int\int (x + xy)dx \wedge dy$$

Integrals become a natural manner to evaluate the flow of a curve along a differential 1-form. Using the hodge star operator we can evaluate naturally the flow through a curve on a surface.

1.4.6.1 Simplifying theorems using differential forms

Stokes theorem:

$$\int_{\Omega} d\alpha = \int_{\partial\Omega} \alpha$$

If we have a differential $(n-1)$-form α on a n-dimensional domain Ω, then the integral over Ω of the exterior derivative $d\alpha$ is equal to the integral of α over the boundary of the domain $\partial\Omega$.

The divergence theorem can be expressed as:

$$\int_{\Omega} \nabla \cdot X \, dA = \int_{\partial\Omega} n \cdot X \, d\ell$$

But equally we can write the divergence using differential forms and stokes theorem.

$$\int_{\Omega} d \star \alpha = \int_{\partial\Omega} \star \alpha$$

Naturally, this describes the internal product with the normal using the hodge star operator, which becomes natural based on the definition of it (the hodge star operator).

Greens theorem can be expressed as:

$$\int_{\Omega} \nabla \times X \, dA = \int_{\partial\Omega} t \cdot X \, d\ell$$

Writing it using differentia forms it becomes

$$\int_{\Omega} d\alpha = \int_{\partial\Omega} \alpha$$

This allows for a more intuitive reading of differential k-forms as we can see as they represent the tangent component of the corresponding vector field X to the surface.

1.4.7 Inner product of differential k- forms

Formally the inner product is defined as any symmetric bilinear map $\langle \cdot , \cdot \rangle : V \times V \to \mathbb{R}$, which has the following properties:

1. $\langle u , v \rangle = \langle v , u \rangle$
2. $\langle u + v, w \rangle = \langle u , w \rangle + \langle v , w \rangle$
3. $\langle au , v \rangle = a\langle u , v \rangle = \langle u , av \rangle$
4. $\langle u , u \rangle \geq 0; \ \langle u , u \rangle = 0 \Leftrightarrow u = 0$

For k-forms, the inner product is defined as: Let $\alpha, \beta \in \Omega^k$ be any two differential k-forms. Their inner product is defined as

$$\langle\langle \alpha, \beta \rangle\rangle \coloneqq \int_{\Omega} \star \alpha \wedge \beta$$

1.5 DISCRETE DIFFERENTIAL FORMS

So far, we have written about the concepts used to build up exterior calculus, which is one of the common languages to treat curved surfaces. Since we work on discrete surfaces (meshes), it's important to mention the particularities they present.

The simplest way to approach discretization of differential forms and their operators is to use "equivalence". Meaning for each term of the continuous setting we presented, we will present an equivalent object in the discrete setting which will act, for the most part, in the same manner.

Domains can be directly transformed into meshes, particularly conforming meshes. And differential forms can directly be represented by values in particular structures of the mesh. Differential k-forms can be directly represented by corresponding structures of k-dimension. And differential operators can be represented by a corresponding sparse matrix.

The process to discretize a differential k-form describes the creation of a *de Rham map.* A map where for every directed structure forming a mesh a relative differential k-form is integrated along it. Similarly, to interpolate from values on a mesh to a continuous setting a linear combination of continuous functions is associated with the directed k-dimensional structures of the mesh, this process is known as *Whitney interpolation.* We may refer to the k-dimensional structures that form meshes as k-simplices (or if singular a k-simplex).

1.5.1 Discretization

For continuous differential 1-forms $\in V^*$, integration along 1-dimensional curves gives us the relative alignment of those curves with the selected 1-form. For a mesh every oriented edge (1-simplex) can be seen as a 1-dimensional curve. We can integrate the original 1-form along every edge of a mesh and the resulting value will be the discrete representation of the original 1-form on the given mesh. Specifically, the normal procedure is: Compute the tangent T to and edge e, evaluate the desired 1-form α for T, getting the function $\alpha(T)$. Finally integrate the resulting scalar function along the edge. Normally a numeric approximation is used and is considered "good" enough to evaluate the result.

$$\widehat{\alpha_e} \coloneqq \int_e \alpha = \int_0^T \alpha(T) ds \approx length(e) \left(\frac{1}{N} \sum_{i=1}^{N} \alpha_{p_i}(T) \right)$$

This can be expanded to k-forms in general. The formal definition is:

Let ω be a differential k-form on $\mathbb{R}^n$, and let K be an oriented simplicial complex[1]. For each k-simplex σ in K, the corresponding value of the discrete k-form is

$$\widehat{\omega}_\sigma := \int_\sigma \omega$$

The map from continuous forms to discrete forms is called the *discretization map* or the *de Rham map.*

In crude words, for any k-form forming a simplicial complex (a mesh) we can calculate the corresponding discrete k-form by integrating on the corresponding k-simpleces.

For practical purposes, this means that values at vertices represent discrete 0-forms, values at edges represent discrete 1-forms and values at faces represent discrete 2-forms.

Now that we have encoded k-forms on our mesh, we need a way to represent them accordingly. The simplest solution is using column vectors, where for each entry on the vector, a corresponding

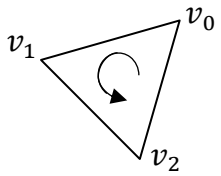


Figure 2 representation of σ

k-simplex exists. Meaning for each existing k-simplex we assign them a unique index from 0 to $(N_k - 1)$, where N_k is the total number of unique elements of k-dimensionality, the only detail to keep in mind is that the change of direction of an l-simplex, where $l > 0$, implies a change of sign of the corresponding k-form values.

1.5.2 Chains and Cochains

A Chain is a linear combination of vector basis σ_i associated to every k-simplex on a mesh. Formally: a *chain group* C_k is a free Abelian group generated by k-simplices.

They are particularly useful when defining sections, segments and boundaries specific to a mesh.

Definition: Let $\sigma := (v_{i_0}, \dots, v_{i_k})$ be an oriented k-simplex. Its *boundary* is the oriented $(k-1)$-chain

[1] Simplicial complex: is a set composed by points, planar polygons and line segments and their corresponding n-dimensional counterparts.

$$\partial\sigma \coloneqq \sum_{p=0}^{k} (-1)^p (v_{i_0}, \dots, \widetilde{v_{i_p}}, \dots, v_{i_k})$$

Where $\widetilde{v_{i_p}}$ indicates that the p vertex has been omitted.

e.g. Consider the 2-simplex $\sigma \coloneqq (v_0, v_1, v_2)$ Figure 2, its boundary is the 1-chain $(v_0, v_1) + (v_1, v_2) - (v_0, v_2)$

The boundary is linear, meaning that the boundary of the sum of chains is equal to the sum of the boundary of different chains (since for different chains the intersecting parts will have opposite signs). i.e.,

$$\partial \sum_i c_i \sigma_i = \sum_i \partial c_i \sigma_i$$

The *coboundary* of an oriented k-simplex σ is the collection of all oriented $(k+1)$-simplices that contain σ, and have the same relative orientation. In other words, for a vertex, is all edges that contain it with direction towards it, for an edge is the adjacent faces where the edge keeps its orientation when describing the faces, this means the coboundary has opposite directions for the two faces next to an edge.

Cochains are the representation of k-forms on a mesh. And are associated as the dual of chains. And can be seen as a function mapping chains to a real number.

1.5.3 Interpolation

Given a simplicial complex K, made exclusively of triangular faces, the *hat function* or *Lagrange basis* ϕ_i is a real-valued function that is linear over each simplex and $\phi_i(v_j) = \delta_{ij}$, meaning that: for each vertex v_j if $i = j$ then it equals 1 and 0 otherwise. Given a discrete 0-form $u: V \to \mathbb{R}$, the interpolating 0-form is given by

$$u(x) = \sum_i u_i \phi_i(x)$$

Meaning we can interpolate linearly using the values between the vertices. Giving us a continuous non-differentiable 0-form. This 0-form at any point p can be expressed by a weighted sum of the three surrounding vertices, where the coefficients weighing the respective vertices are the corresponding barycentric coordinates of the point p in the 2-simplex t.

For 1-forms a similar concept can be applied. The *Whitney 1-forms* are differential 1-forms associated with each edge ij, given by

$$\phi_{ij} \coloneqq \phi_i d\phi_j - \phi_j d\phi_i$$

The Whitney 1-forms can be used to interpolate a discrete 1-form $\widehat{\omega}$ via

$$\sum_{ij} \widehat{\omega}_{ij} \phi_{ij}$$

More generally the *Whitney k-form* is associated with the oriented k-simplex $(i_0, .., i_k)$ and given by

$$\sum_{p=0}^{k} (-1)^p \phi_{I_p} d\phi_{i_0} \wedge \ldots \wedge \widehat{d\phi_{\iota_p}} \wedge d\phi_{i_k}$$

Where $\widehat{d\phi_{\iota_p}}$ is omitted.

1.5.4 Discrete exterior derivative

To get the discrete exterior derivative is like applying the exterior derivative to a continuous k-form and integrating the result over oriented simplices. We can exemplify this using a 0-form. Given the discrete 0-form $\hat{\phi}$ on the vertices v_1, v_2; its discrete derivative is the discrete 1-form $\widehat{d\phi}$. $\widehat{d\phi}$ can be exactly calculated along the edge e knowing the values of the 1-form at v_1, v_2 (which are the boundary of e). By definition, the value of $\widehat{d\phi}$ along e is

$$\left(\widehat{d\phi}\right)_e = \int_e d\phi$$

Using stokes theorem we can rewrite this as

$$\left(\widehat{d\phi}\right)_e = \int_{\partial e} \phi \; = \widehat{\phi_2} - \widehat{\phi_1}$$

Meaning we can use the fundamental theorem of calculus we can precisely calculate $\widehat{d\phi}$ along e by just knowing the values at the vertices which limit it.

The same process can be applied to discrete 1-forms to get their derivatives. Given the discrete differential 1-form $\hat{\alpha}$ along some oriented edges $\{e_1, \ldots, e_{\not f}\}$ defining the boundary of an oriented face f , to get the primal 2-form $\widehat{d\alpha}$ on f we can do a similar process as before.

$$(\widehat{d\alpha})_f = \int_f d\alpha = \int_{\partial f} \alpha$$

The boundary of f is the set $\{e_1, \ldots, e_f\}$, meaning we can rewrite the previous expression as

$$\int_{\partial f} \alpha = \sum_{i=1}^{\#} \int_{e_i} \alpha = \sum_{i=1}^{\#} \hat{\alpha}_i$$

(note that we assume that the edges have the same direction as f else all edges with contrary sense would be negative in value).

This all just implies that the discrete exterior derivative is the same as applying the coboundary operator to the corresponding cochain of a simplex.

By this definition, the exterior derivative is a purely topological operator, meaning is only related to the connectivity of the studied mesh and the relative shape it has is irrelevant.

1.5.5 Dual discrete differential k-forms

A dual discrete differential k-form is a k-form which is defined and has a value per dual k-cell. Meaning a dual 0-form has a value per each dual vertex, a dual 1-form has a value per each dual edge and a dual 2-form has a value per each dual cell. (N.B. The existence of dual and primal forms is a phenomenon exclusive to the discrete setting).

Similarly, as in the primal case, the dual exterior derivative brings a k-form to a $(k+1)$-form. The values of $(k+1)$-form can be obtained by summing up the "oriented" values of the desired k-form at the boundary of the respective $(k+1)$-cell.

They tend to be hard to interpolate and the Whitney bases approach does not work, since in general k-cells tend to be N-gons and tend to be non-planar. This leads to the suggestion to use a different kind of basis for interpolation, but there is no standard, and the selection of the basis depends on the reason of why the interpolation is being done.

1.5.6 Discrete Hodge star

The discrete hodge star or *diagonal hodge star*, when working on the discrete setting on an n-dimensional space, will map directly all k-simplices to the corresponding dual $(n-k)$-cell. Meaning, for our purposes, it will directly apply the transformations explained in 1.3. It would be more accurate to do so if we selected the circumcenter instead of the barycenter to define the dual k-cells, but due to the possibility of getting negative values for obtuse triangles it was decided that the barycenter was a better choice.

Let Ω_k and $\Omega^{\star}_{n-k}$ be a primal k-form and its dual $(n-k)$-cell respectively. The discrete hodge is a map $\star\colon \Omega_k \to \Omega^{\star}_{n-k}$ determined by

$$\widehat{\star\alpha}(\sigma^{\star}) = \frac{|\sigma^{\star}|}{|\sigma|}\hat{\alpha}(\sigma)$$

for each k-simplex σ in our n-dimensional simplicial manifold, where $\sigma^{\star}$is the corresponding dual cell and $|\cdot|$ gives the length or volume of said simplex/cell (for vertices this value will be 1).

1.5.7 Vector valued differential forms

A vector valued k-form is a fully antisymmetric multi-linear map from k vectors in a vector space V to another vector space U.

Most operations work similarly as they do for regular k-forms, except they are applied component wise to each vector component. One important exception is the wedge product. To evaluate the wedge product, it's necessary to use a product. Since our previous definition was working with scalar values regular multiplication was enough, but since now we are working with n-dimensional vectors we need to be more specific. We mostly will work with mappings from $\mathbb{R}^3 \to \mathbb{R}^3$ a natural choice is to use the cross product. This is justified when working on curved surfaces as the result of these operations are useful quantities with respect to the studied curves. Importantly unlike with the regular product a special property changes when working using the cross product. Specifically, when using normal multiplication both the order of the operands and the order of the operators are anti-symmetric. But, since the cross product is anti-symmetric itself, its change of sign cancels out the anti-symmetry if operators swap. In other words

$$(\beta \wedge \alpha)(u, v) = \beta(u) \times \alpha(v) - \beta(v) \times \alpha(u)$$
$$= \alpha(u) \times \beta(v) - \alpha(v) \times \beta(u)$$

Meaning

$$\beta \wedge \alpha = \alpha \wedge \beta$$

This also implies that when we wedge a $(\mathbb{R}^3, x)$-valued 1-form with itself it will have a result different from 0.

$$(\alpha \wedge \alpha)(u, v) = \alpha(v) \times \alpha(u) - \alpha(u) \times \alpha(v) = 2\,\alpha(u) \times \alpha(v) \neq 0$$

A vector valued differential k-form is a vector valued k-form defined at every point in a domain.

1.6 CURVES AND SURFACES

Now that we have defined the most relevant operators to work on meshes, we will specify the definitions for curves and surfaces. This will allow us to more precisely work on meshes, which are discrete surfaces and discrete curves.

We may refer to curves as 1-dimensional manifolds and to surfaces as 2-dimensional ones.

1.6.1 Smooth curves

A parametrized plane curve is a map taking each point in an interval $[0, L]$ of the real line to some point in $\mathbb{R}^2$ ($\gamma: [0, L] \to \mathbb{R}^2$), we will assume curves to be as differentiable as needed in this section.

We can think of a planar curve as an $\mathbb{R}^2$-valued 0-form on an interval of the real line, the exterior derivative of this curve is mapped into the plane at each point.

Any curve $\gamma: [0, L] \to \mathbb{R}^2$ can be *reparametrized* by applying a bijection $\eta: [0, L], \to [0, L]$. Obtaining a new parametrized curve $\tilde{\gamma}(t) \coloneqq \gamma(\eta(t))$. the differentiation of $\tilde{\gamma}$ is $d\tilde{\gamma} = d\gamma \circ d\eta$.

A curve is embedded if it's continuous and a bijective map from its domain to its image, and the inverse map is also continuous.

The curvature of an arc-length parametrized plane curve is the rate of change of the tangent. More precisely:

$$\kappa(s) \coloneqq \langle N(s), \frac{d}{ds} T(s) \rangle = \langle N(s), \frac{d^2}{ds^2} \gamma(s) \rangle$$

Where k is the curvature of the curve, N is the unit normal and T is the unit tangent of the curve at the point s.

An equivalent way to define curvature for smooth surfaces is with respect to a specific direction,

$$\kappa(s) = \frac{d}{ds} \theta(s)$$

Where θ is the angle between the tangent and the preselected direction.

An equivalent way to define curvature is using the so-called *Osculating circle.* The osculating circle is the circle that best approximates the curve at any given point p with some conditions. If we consider two neighboring points to p, to either side of it. The osculating circle is the circle passing through all three points as these neighbors approach p. The curvature of p will be the reciprocal of the radius r of the osculating circle $\kappa(p) = \frac{1}{r(p)}$.

These definitions are important to characterize curves as per the *fundamental theorem of plane curves*, up to rigid motions (translations and rotations), an arc-length curve is uniquely determined by its curvature. More importantly for us, these different definitions provide us with different results when working in a discrete setting.

For 3-dimensional curves, we also have torsion, which describes the "out of plane bending". To capture this quantity a frame that moves along the curve is used, this frame is called *Frenet frame.* The Frenet frame depends only on the local geometry of the curve. This frame is composed of the unit tangent T (found differentiating the curve at the point p), the unit normal N (found by differentiating T) and the binormal B (orthonormal complement to the plane defined by T, N). The

curvature κ and the torsion τ can be defined in terms of the change of the Frenet frame as we move along the curve.

$$\frac{d}{ds}\begin{bmatrix}T\\N\\B\end{bmatrix}=\begin{bmatrix}0 & -\kappa & 0\\ \kappa & 0 & -\tau\\ 0 & \tau & 0\end{bmatrix}\begin{bmatrix}T\\N\\B\end{bmatrix}$$

Meaning, the change in T is the bending of the curve (*curvature*) and the change in B describes twisting (*torsion*). So, in 3D

$$\kappa = -\langle N, \frac{d}{ds}T\rangle$$

$$\tau = \langle N, \frac{d}{ds}B\rangle$$

The *fundamental theorem of space curves* says that given curvature and torsion of an arc-length parametrized space curve, we can recover the curve itself (up to rigid motions).

Frenet frames are not defined along straight segments of curves, but in those cases an *adapted Frenet frame* is used. An adapted Frenet frame is any orthonormal frame containing the unit tangent.

1.6.1.1 Turning and winding numbers

The **turning number k** is the number of counterclockwise turns made by the tangent for a single "walk" on the curve. It can be thought of as the tangent map of the unit circle. k is the *topological degree* of this map. Meaning the number of times the tangent covers the circle as we go around the curve.

$$T(s)=\frac{\gamma'(s)}{|\gamma'(s)|} \qquad k = degree(T(s))$$

The **winding number n** is the number of times a curve "encases" a point. Or more precisely, is the total signed length of the projection of the curve onto a unit circle around p. Considering a map $\hat{\gamma}_p(s)$ constructed by projecting the curve onto the unit circle around p. n is the topological degree of the map.

$$\hat{\gamma}_p(s) \coloneqq \frac{\gamma(s)-p}{|\gamma(s)-p|} \qquad n = degree(\hat{\gamma}_p(s))$$

The winding number is particularly useful to determine if a mesh is watertight and the location of points with respect to the mesh (inside or outside).

1.6.2 Discrete curves

A Discrete curve is a *piecewise linear* parametrized curve. In simple terms: a sequence of points γ_i connected by straight line segments. For simplicity given the total curve γ the segment

connecting the points γ_i and γ_j will be called γ_{ij}, where i is the index of the points discretizing the curve, and j the index of the following one. As a differential form, discrete curves are described as $\mathbb{R}^n$-valued 0-forms γ on a simplicial complex.

The discrete differential of γ_{ij} can be calculated simply as

$$(d\gamma)_{ij} = |\gamma_j - \gamma_i|$$

The unit tangent is

$$T_{ij} \coloneqq \frac{(d\gamma)_{ij}}{|(d\gamma)_{ij}|}$$

Discrete curves are hard to define as regular as they can have degenerate behaviors not commonly seen in smooth curves. For starters the vertices are not well defined when describing the differential of the curve. Many definitions that follow the smooth setting definition tend to have problems capturing all the properties desired from a smooth curve (as, for example, avoiding degenerate segments or angles).

As an example we will give two valid definitions:

- First, *a discrete curve is defined discretely regular if its discrete differential is nonzero for all its segments*. This definition prevents degenerate segment lengths, but it fails to prevent zero angles. But note that some applications do use this definition.
- A second definition of regular discrete curve is: *a discrete curve is discretely regular if it is a locally injective map.* This second definition is better as it does not depend on differentiation and captures the desired properties of the smooth setting in a better way (non-zero angles and non-zero length edges) but is harder to calculate in general.

Discrete space curves can be uniquely defined, with a difference of rigid motions, by their edge lengths, their curvatures (for now think of them as a function of the exterior angle of any one specific vertex with respect to its previous connection on the curve) and the torsion. Torsion is defined using three consecutive edges $\gamma_{i,i+1}, \gamma_{i+1,i+2}, \gamma_{i+2,i+3}$. Where the first two edges define a plane and the last two define a second plane, change of the angles between those planes is the torsion. So, torsion is associated with edges, specifically to the middle edge when doing the calculation.

1.6.2.1 Discrete curvature

Previously in 1.6.1 we defined in several forms what smooth curvature is, unfortunately this concept in the discrete domain is particularly hard to translate. Using different definitions, which are equivalent in the smooth setting, can provide different results when translated to the discrete setting. These results tend to capture different properties in different ways. In general, a "good" definition of discrete curvature:

- Will provide some specific properties that we wish to emulate from the continuous setting.
- Will converge to the same value as the smooth setting as the curve becomes more refined.

- Is efficient to compute or calculate.

Using different definitions for curvature in the smooth setting, we can create equivalent discrete definitions.

Turning angle: if the initial definition of curvature is the rate of change of the angle of the tangent with the horizontal direction $\left(\kappa(s) = \frac{d}{ds}\theta(s)\right)$. This means that κ, in a sufficiently small segment $[a, b]$, can be written as

$$\int_a^b \kappa(s)ds = \int_a^b \frac{d}{ds}\theta(s)ds = \theta(b) - \theta(a)$$

In the discrete setting, this means we can directly use θ for the curvature of straight segments γ and calculate the curvature at vertices as the difference between $\theta_{i-1,i}$ and $\theta_{i,i+1}$, this is the exterior angle at the vertex i and will be called θ_i for the future examples. This curvature will be denominated κ_i^A at vertex i.

Length variation: In the smooth setting the fastest way to decrease the length of a curve is to move it in the normal direction, with speed proportional to κ. Formally, considering an *arbitrary* change in the curve γ, given by the function $\eta: [0, L] \to \mathbb{R}^2$ with $\eta(0) = \eta(L) = 0$. Then $\frac{d}{d\varepsilon}\Big|_{\varepsilon=0} length(\gamma + \varepsilon\eta) = -\int_0^L \langle \eta(s), \kappa(s)N(s)\rangle\, ds$, the motion with the quickest decrease in length is $\eta = \kappa N$.

In the discrete setting, given the segment $[a, b]$, its length $\ell := |b - a|$, the motion that provides the fastest increase in length is the motion along the b direction (or the a direction), the unit vector proving that direction is $\nabla_b \ell = \frac{b-a}{\ell}$. For a vertex i the discrete curvature on it will be the sum of the contributing increase of the connecting segments. Meaning we can calculate $\nabla_{v_i} L = 2\sin\left(\frac{\theta_i}{2}\right) N_i$ where L is the sum of the length of both adjacent segments and θ_i is the exterior angle at the vertex, and N_i is the normalized sum of both tangent vectors to the adjacent sides of v_i (this value N_i can be seen as a discrete normal for v_i). Therefore

$$\kappa_i^B N_i := 2\sin\left(\frac{\theta_i}{2}\right) N_i$$

$$\kappa_i^B = 2\sin\left(\frac{\theta_i}{2}\right)$$

Steiner Formula: Steiner's formula says that if we move at a constant velocity in the normal direction, the change of length of the curve is proportional to its curvature.

$$length(\gamma + \varepsilon N) = length(\gamma) - \varepsilon \int_0^L \kappa(s)ds$$

In the discrete setting without well-defined normals on the vertices, the best first step is to calculate this operation is to translate edges in their normal directions. This will create disconnected segments separated from their original positions. To complete the operation, it is needed to reconnect the segments. It normally is done with one of three methods, either extend the segments at an equal rate until they collide in the middle, or we can create a new segment connecting previously adjacent segments on their previously shared vertex, or in those same vertices connect them using a circular arc of radius ε. This provides three different length variations:

$$extension\ of\ previous\ curve\ length = length(\gamma) - \varepsilon \sum_i 2\tan\left(\frac{\theta_i}{2}\right)$$

$$straight\ segment\ length = length(\gamma) - \varepsilon \sum_i 2\sin\left(\frac{\theta_i}{2}\right)$$

$$Circle\ connection\ length = \ length(\gamma) - \varepsilon \sum_i \theta_i$$

These three methods provide three valid definitions for discrete curvature

$$\kappa_i^A \coloneqq \theta_i$$

$$\kappa_i^B \coloneqq 2\sin\left(\frac{\theta_i}{2}\right)$$

$$\kappa_i^C \coloneqq 2\tan\left(\frac{\theta_i}{2}\right)$$

Osculating circle: as previously stated, curvature can be described by an osculating circle that passes through three ordered points, where the middle point is the one we wish to evaluate. This in the discrete setting can be done using the two adjacent vertices in the curve to the evaluated vertex v_i.

$$\kappa_i^D \coloneqq \frac{1}{r_i} = \frac{2\sin\theta_i}{\omega_i}$$

Where ω_i is the direct distance between the vertices v_{i-1} and v_{i+1}.

These four different definitions for discrete curvature are equally valid and are normally chosen depending on the problem at hand.

1.6.2.2 Curvature flow on curves

A curvature flow is a time evolution of a curve (or surface) driven by its curvature.

This concept is used to optimize some energy function which can provide some desired result. For example, a common use for curvature flow is to find geodesics on a surface.

A use that can be applied to this concept is using machine learning, where given the optimization energy, a neural network model can optimize iteratively on a same curve, to approximate the result numerically.

1.6.3 Smooth surfaces

In the following section we will provide the basis to talk about discrete surfaces (meshes), specifically we will provide the insight to compare them with smooth surfaces and the different approaches there exist to describe them.

We can describe how a surface sits in space (describing it as a function of its coordinates); this is called an extrinsic representation. If we describe a small patch of said surface, we are describing it locally. If we then can join the total of local patches into a single representation, then we have a global representation. A second option when "joining" these patches is to directly map from one "patch" to another, avoiding general space embeddings, and all the geometric information of our surface can be recovered using the *Riemannian metric.* This last part is an intrinsic description, meaning that the surface can be described purely by its geometry independent of any kind of embedding. For our purposes we can think of both these approaches as describing a mesh via its vertex positions (extrinsically) or via its edge lengths (intrinsically).

1.6.3.1 Parametrized surfaces

A parametrized surface is a map $f: U \to \mathbb{R}^n$ from a two-dimensional region $U \subset \mathbb{R}^2$ into space.

The set $f(U)$ is called the image of f, and describes a subset of $\mathbb{R}^n$.

A parametrized surface f is an *embedding* if it's a continuous bijection onto its image $f(U)$, with continuous inverse. Meaning that the topology of the parametrized surface is maintained by its parametrization.

The differential of f can be written as

$$df = \sum_i \frac{\partial f}{\partial x_i} dx_i$$

Where dx_i are basis 1-forms and $\frac{\partial f}{\partial x_i}$ are the partial derivatives of f with respect to the basis directions. This is the same as the *Jacobian (J_f)* of f, meaning $df(X) = J_f X$ where X is a vector field in the original parametrized domain.

A surface will be called *immersion* if its differential is nondegenerate.

$$df(X)|_p = 0 \iff X|_p = 0 \ \ \forall p \in U$$

A *Regular homotopy* is a transformation of a surface that doesn't involve pinches, creases or stops. Formally: given two mappings $f_0, f_1 : U \to \mathbb{R}^3$, a regular domain homotopy will be a continuous map $h: U \times [0,1] \to \mathbb{R}^3$. $h(x,0) = f_0(x)$ and $h(x,1) = f_1(x)$, and $h(\cdot, t)$ is an immersion for all

values of t. Meaning that the transformation from f_1 to f_0 is differentiable at all points of the transition.

1.6.4 The Riemannian metric

The Riemannian metric or induced metric $g(X, \Upsilon)$ (where X and Υ are tangent vectors to the surface) represents the inner product of $\langle X, \Upsilon \rangle$, where the inner product operator can change from point to point. The Riemannian metric is a smoothly varying positive-definite bilinear form.

When expressing a surface using different parametrizations, it is necessary that the result of inner products on them is the same, which if we use classical inner product, is not always the case. We can think of it as making sure that the vectors X, Υ are on a surface by applying $df(X)$ and $df(\Upsilon)$ in $\mathbb{R}^3$, when represented in different parametrizations $f, \tilde{f}$ they need to give the same result of their inner product. The solution is "bringing" the vectors to $\mathbb{R}^3$ and then do the inner product. This is the induced Riemannian metric

$$g(X, \Upsilon) \coloneqq \langle df(X), df(\Upsilon) \rangle$$

This metric can be represented by a 2x2 matrix **I** called the *first fundamental form.*

$$g(X, \Upsilon) = X^T \boldsymbol{I}\, \Upsilon$$

The ij^{th} entry of **I** can be calculated by applying the metric to basis vectors $\frac{\partial}{\partial x^i}$ and $\frac{\partial}{\partial x^j}$,

$$\boldsymbol{I}_{ij} = \langle df\left(\frac{\partial}{\partial x^i}\right), df\left(\frac{\partial}{\partial x^j}\right) \rangle$$

The Riemannian metric can be written in terms of the Jacobian as

$$g(X, \Upsilon) = \left(J_f X\right)^T \left(J_f \Upsilon\right) = X^T \left(J_f^T J_f\right) \Upsilon$$

$$\boldsymbol{I} = J_f^T J_f$$

Importantly this metric varies depending on which point is evaluated. Thinking about as different parts of the surface that are "stretch" differently for each point to fit the parametrization is an easy way to think about it.

1.6.5 Conformal coordinates

When working on surfaces, it is not always possible to have isometric parametrizations, but is always possible to have a parametrization which preserves angles. This kind of angle preserving parametrization is called *conformal*. A parametrization f is conformal if at each point the induced metric is simply a positive rescaling of the 2D Euclidean metric (the identity matrix).

1.6.6 Gauss map

A vector is normal to a surface if it's orthogonal to all tangent vectors ($\forall X, \langle N, df(X)\rangle = 0$). The *Gauss map* is a continuous map associating each point on the surface with a unit normal vector. Surfaces which do not admit a Gauss map (globally) are called non-orientable, an example is the mobius band. An easy way to think about it is: visualizing the Gauss map as a map from the original parametrization to a unit sphere, if that map is not possible then the surface will be non-orientable.

For embedded immersed closed surfaces, the Gauss map is surjective, meaning that for any unit vector v there exists a point on the surface with a normal N, where $N = v$. Hilbert proofs this by constructing a plane normal to v, that plane if translated in direction -v towards the surface, eventually the plane will collide with the surface. At the moment the plane collides with the surface it will be tangent to it, meaning that the normal v of the plane and the surface normal N are parallel.

We can calculate the "average normal" of a patch of surface Ω using the Gauss map.

$$avgN = \frac{1}{area(\Omega)} \int_{\Omega} NdA$$

The integrand NdA is denoted the *vector area* (a vector-valued 2-form). The vector area can easily be expressed with the following relationship

$$df \wedge df(X, \Upsilon) = df(X) \times df(\Upsilon) - df(\Upsilon)df(X) = 2\big(df(X) \times df(\Upsilon)\big) = 2NdA(X, \Upsilon)$$

Therefore

$$\mathcal{A} = \frac{1}{2} df \wedge df = NdA$$

Using this concept on discrete N-gons, we can assign a normal to non-planar polygons for meshes. Since

$$2\int_{\Omega} NdA = \int_{\Omega} df \wedge df = \int_{\partial\Omega} fdf = \int_{\partial\Omega} f(s) \times df(T(s))ds$$

Meaning the average normal of a patch of surface is the same for surfaces with the same boundary, independent of the surface geometry. Meaning for our non-planar N-gons we can assign them a normal using the average normal of a known surface with the same boundary.

1.6.7 Exterior calculus on immersed surfaces

Using exterior calculus on curves spaces we can split topology and geometry depending on the operations we use. Topological operations would be the wedge product or the exterior derivative, while geometry will be described by the hodge star operator.

For a surface immersed in 3D space we need two pieces of data to operate on it: the **area form** ("how big a region is") and the **complex structure** (is a structure that describes $\frac{\pi}{2}$ rotations on the surface). Both these quantities are determined by the induced metric of the surface.

1.6.7.1 The area form

The area form of a surface is described using the area vector.

$$2NdA(X, \Upsilon) = df \wedge df(X, \Upsilon)$$

Since dA is the area 2-form on $f(M)$, we can write it as

$$dA = \frac{1}{2}\langle N, df \wedge df \rangle$$

So given this area 2-forn dA and a scalar function ϕ, we can define the Hodge star on 0-forms as:

$$\phi \overset{\star}{\rightarrow} \phi dA$$

Meaning the original surface has been scaled for each value of ϕ at each point. Equivalently, given the 2-form ω on $f(M)$. The Hodge dual is the unique 0-form Φ such that $\omega = \Phi dA$.

1.6.7.2 The complex structure

The *complex structure* or the *linear complex structure* $\mathcal{J}$ for a surface immersed in $\mathbb{R}^3$ f, describes a $\frac{\pi}{2}$ rotation via a cross product with the unit normal N:

$$df\left(\mathcal{J}_f X\right) \coloneqq N \times df(X)$$

(Note: X may be only perpendicular on the surface described by f to $\mathcal{J}_f$).

Explicitly computing $\mathcal{J}$ can be done by solving a matrix equation.

$$\widehat{N} \coloneqq \begin{bmatrix} 0 & -N_z & N_y \\ N_z & 0 & -N_x \\ -N_y & N_x & 0 \end{bmatrix}$$

Where $\widehat{N}$ is defined by the cross product of the normal N with a vector u

$$N \times u = \widehat{N}u$$

The differential will be represented as the Jacobian matrix A

$$A \coloneqq \begin{bmatrix} \partial f_x/\partial u & \partial f_x/\partial v \\ \partial f_y/\partial u & \partial f_y/\partial v \\ \partial f_z/\partial u & \partial f_z/\partial v \end{bmatrix}$$

And our complex structure is the 2×2 matrix J

$$J \coloneqq \begin{bmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{bmatrix}$$

The we can calculate J as

$$J = (A^T A)^{-1}(A^T \widehat{N} A)$$

This allows us to define the *induced Hodge star* on 1-forms. For a 1-form α in a plane, the application of $\star\,\alpha$ to a vector X is defined as a $\frac{\pi}{2}$ rotation of X in counterclockwise direction. For an immersed surface f, we want this rotation to be with respect to the surface itself.

$$\star_f \alpha(X) = \alpha(\mathcal{J}_f X)$$

1.6.7.3 The metric calculated with the area form and the complex structure

We can use the previously defined area form dA and the complex structure to explicitly calculate the Riemannian metric.

$$g(X, \Upsilon) = dA(X, J\Upsilon)$$

A simpler visualization would be to use the planar equivalents of these operators. Meaning the same expression in the plane can be written as

$$|X \cdot \Upsilon| = |X \times (J\Upsilon)|$$

Using the Riemannian metric, we can explicitly define the # and the ♭ operators for non-planar surfaces.

$$X^{\flat}(\Upsilon) \coloneqq g(X, \Upsilon)$$

$$g(\alpha^{\#}, \Upsilon) \coloneqq \alpha(\Upsilon)$$

Unlike in Euclidean space, this operation no longer can be seen as a "transpose".

1.7 DISCRETE SURFACES

When studying discrete surfaces two main models are presented. Simplicial 2-manifolds or trimeshes, which naturally allow for a nice application of discrete exterior calculus. And the second model are Nets or quad-meshes, which are piecewise integer lattices.

For our discrete surfaces require two main regularity conditions, first the surface needs to be *manifold,* and the vertex coordinates should describe a *simplicial immersion*.

A simplicial surface is a manifold simplicial 2-complex, normally (and for this chapter) are constructed purely by triangles, their edges are contained at most in two triangles and minimum in one if they are a boundary. Every vertex belongs to one and only one single edge connected cycle of triangles or path along the boundary. We will call these simplicial surfaces as $K = (V, E, F)$ where V is the set of vertices forming it, E the set of edges and F the set of faces (in this set the

vertices are purely indices, not coordinates). Meaning K is a complex describing the topology of the surface, but it does not have information on its geometry.

To capture the geometry of the surface we will assign coordinates f_i to each vertex (discrete $\mathbb{R}^n$-valued 0-form). The edges and the faces will be described by a linear interpolation of these coordinates, using barycentric coordinates. f is a simplicial map (a map from a simplices to simplices) from $K \to \mathbb{R}^n$.

As previously mentioned in 1.5, we can simply describe the differential of our map by integrating along the desired evaluations. Meaning that $(df)_{ij} \coloneqq f_j - f_i$ the differential of the edge e_{ij} is just the difference in the values at each vertex f_i and f_j. $(df)_{ij}$ is still anti-symmetric.

In the smooth setting an immersion was defined if the differential was non-degenerate. In the discrete setting we need to take care specifically of branch points, which are degenerate vertices whose behavior is not captured by the discrete differential. Therefore, this provides the need to use a different definition for discrete immersion. The definition used for discrete immersion is done using local injectivity. Discrete immersion is a locally injective simplicial map for all parts of the mesh.

1.7.1 Discrete Gauss map

For a discrete immersion, the Gauss map is the unit normals of the faces of K. The discrete Gauss map is a dual discrete $\mathbb{R}^3$-valued 0-form (vector per triangle). With this we can plot the normal in the unit sphere. Connecting adjacent normals with arcs (on the sphere) we get a family of normal orthogonal to the corresponding edge. Similarly for vertices, we can describe their normal by the area created by the polygon with vertices on the face normals projected on the unit sphere.

Selecting a specific normal for both edges and vertices depends on application, but there is not a "correct" choice. Some definitions, although intuitive, may behave poorly on given environments. A good definition can be the weighted sum of the normals of the surroundings faces, weighed by their area over the total area around the vertex. A second option is to calculate the normal using the dual cell of a vertex and calculate its average normal. For example, for a hexagon cell, this would be

$$\frac{1}{3}\int_{\Omega} N dA = \frac{1}{6}\int_{\partial\Omega} f \times df = \frac{1}{6}\sum_{ij\in\partial\Omega}\int_{e_{ij}} f \times df = \frac{1}{6}\sum_{ij\in\partial\Omega}\frac{(f_i + f_j)}{2} \times (f_j - f_i) = \frac{1}{6}\sum_{ij\in\partial\Omega} f_i \times f_j$$

This definition is independent of the position of the initial vertex. A third possible definition is to use an angle weighed sum of the normals. Much like the area weighted, this sum would be weighted on the angles around the vertex i.

1.7.2 Discrete exterior calculus on discrete surfaces

The previously discussed definition for discrete hodge start or diagonal hodge star, still work for

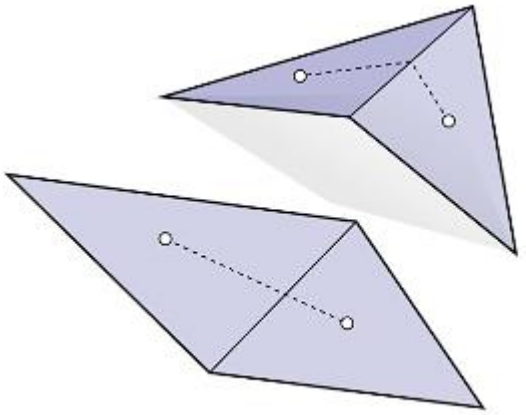


Figure 3 representation of ⋆ of 1-form on a curved discrete surface [1]

discrete 3D surfaces but we must keep in mind that the measurements are dependent on the "curvature" or bending between the faces.

Meaning that for our 1-form we need to consider the bent edge going from one barycenter to the other over the surface (Figure 3), using the $cotan$ formula we can calculate the predefined ratio without having to worry about the bending, as it depends on the opposing angles.

For 0-/2- forms we need to consider the total area as the value (this area may overlap itself if flattened on a plane, e.g. a concave cone), luckily for this step the sum of the small areas at each triangle is a good definition and is simple to calculate. It is not necessary to "unbend" the surface to work on it as the area of the triangles is independent of the bending of the surface.

On discrete curved surfaces we can describe the Laplacian as the *Laplace-Beltrami* operator. It is defined as:

$$\Delta\phi = \star\, d \star d\phi$$

We are going to talk more about it on 1.8.1. For now, we can write it for 1-forms using the $cotan$ formula as:

$$(\Delta u)_i = \frac{1}{2}\sum_{ij\in E}(cot\alpha_{ij} + cot\beta_{ij})(u_j - u_I)$$

1.7.3 Recovering discrete surfaces

For a simplicial immersed closed surface, we can recover its shape uniquely by its connectivity and its gauss map. Following an iterative algorithm we can recover the entire trimesh. The algorithm is as follows:

- Cross product of two adjacent normals gives the middle edge direction.
- The angles between edges can be calculated by the dot product of two adjacent edges.
    - This uniquely determines all triangles on a mesh up to a scale factor (the scale is selected so triangles match adjacent faces).

Repeating the precious steps, we can reconstruct all triangles of a mesh and since we know connectivity, we can completely reconstruct the figure up to a global scaling factor and a global translation factor.

This procedure is only possible in the discrete setting for simplicial surfaces since triangles are determined by their inner angles.

In general, to recover any smooth convex surface is sufficient to have its Riemannian metric. In the discrete setting a similar theorem says that a convex polyhedron can be uniquely determined by its edge lengths.

1.7.4 Curvature in surfaces

Given a curve γ embedded in a surface $f(M)$ it can be described as two distinct curvatures. *Normal* curvature κ_n and *geodesic* curvature κ_g. For curves on a surface, we use a frame based on the tangent of the curve T, and the normal of the surface N_M, the final component of the frame will be $B_M \coloneqq T \times N_M$.

$$\kappa_n \coloneqq \langle N_M, \frac{d}{ds} T \rangle$$

$$\kappa_g \coloneqq \langle B_M, \frac{d}{ds} T \rangle$$

κ_g describes how "straight" a curve is, meaning it determines how much a curve oscillates on a given surface without leaving the surface. If $\kappa_g = 0$, it means the curve is going in a straight path along the surface (is a geodesic). κ_n defines the curvature of γ with relation to space, an example of a curve with $k_g = 0$ but a large κ_n would be a meridian on the globe.

1.7.4.1 Weingarten Map and principal curvature

The Weingarten map dN is the differential of the Gauss map N. At any point $dN(X)$ gives the change in the normal vector along a given direction X. Since the change in a unit normal can't have a component in the normal direction, this means that dN is always tangent to the surface. Much like the Gauss map could be thought as a map to a unit sphere, the Weingart map is tangent to the unit sphere. If we observe the environment around a point p on the surface, let it be the circle formed by all points within a distance r of p, when mapped the normals of all the observed points to the unit sphere, they will take the form of an ellipse. The axis of the ellipse, created by the projection of the normals around p, are the *principal curvature directions* at p. The *principal curvatures* or *principal curvature directions* are two perpendicular directions (in $\mathbb{R}^3$) tangent to the surface on p where the surface has the most and the least curvature for p. The principal

curvatures are normally denoted as κ_1, κ_2 (these correspond to the directions and values of the radii of the ellipse previously described).

1.7.4.2 Normal curvature

Normal curvature is the rate at which the normal is "bending" along a given tangent direction.

$$\kappa_N(X) \coloneqq \frac{\langle df(X), dN(X) \rangle}{|df(X)|^2}$$

It can be seen as the curvature of the curve resulting from the intersection of the surface and the plane described by N and X at a point p (from where we selected the tangent vector X).

Along the possible directions X the one with the maximum and minimum values are the *principal directions (X_1, X_2)*, and their corresponding curvatures are the principal curvatures. The principal directions are perpendicular in $\mathbb{R}^3$, meaning $g(X_1, X_2) = 0$.

When applying the principal directions to the Weingart map, the result is the principal curvature along the principal direction itself (similar as how eigenvectors and eigen values are related).

$$dN(X_i) = \kappa_i df(X_i)$$

1.7.4.2.1 Shape operator

The **shape operator** is a linear map S from tangent vectors to tangent vectors, such that

$$df(SX) = dN(X)$$

The eigenvectors of S are the principal directions, and the principal curvatures are the eigenvalues of S. S is not a symmetric matrix, meaning that its eigenvectors are not orthogonal in $\mathbb{R}^2$, but they will be orthogonal with respect to the Riemannian metric g.

When both our principal curvatures are equal ($\kappa_1 = \kappa_2$), that means that we are at an *umbilic point.* This means that principal directions are not uniquely determined, like in a sphere. S in these points has repeated eigenvalues.

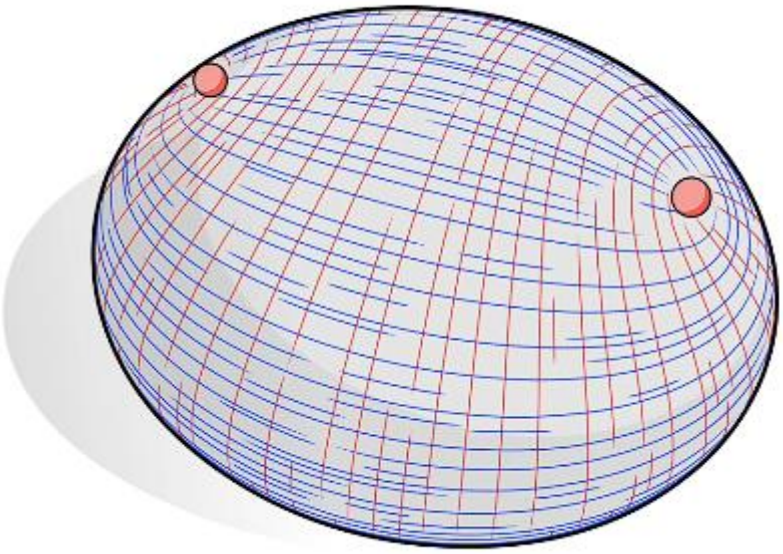


Figure 4 Example of principal curvature lines on a surface, the red lines are the maximum curvature, the blue ones are the minimum curvature, and the two dots are umbilical points on the surface. [1]

The principal directions can be traced creating principal curvature lines. And since they are orthogonal along all the surface, they are very commonly used to create curvature nets on the surface (Figure 4). The complete set of all curvature lines is called the principal curvature network. Principal curvature networks are commonly used for quad-meshing algorithms.

1.7.4.2.2 Separatrices and spirals

If all principal curvature lines end up merging at umbilic points it would be possible to use them directly for meshing, if they can be traced from one umbilic to another umbilic they are called *separatrices*. But, more often than not, they tend to not align nicely and following the principal curvature directions tends to create spirals on the surface. Many algorithms follow this initial idea and create special procedures to avoid spiraling. A classic example is the creation of motorcycle graphs.

1.7.4.3 Gaussian and mean curvature

Gaussian and *mean* curvature describe local bending of a surface and are described by its principal directions.

$$\boldsymbol{Gaussian}\colon\ K \coloneqq \kappa_1\kappa_2$$

$$\boldsymbol{mean}\colon\ H \coloneqq \frac{1}{2}(\kappa_1 + \kappa_2)$$

(for the mean curvature is a common convention to omit the $\frac{1}{2}$ in the calculation).

The Gaussian curvature describes the "convexity" of a shape, meaning surfaces with positive Gaussian curvature are convex, with zero Gaussian curvature are "developable" and with negative are "saddle like". The mean curvature describes the "amount" of curvature of the surface, when a surface has zero mean curvature it is said that it has *minimal* curvature.

These two quantities allow us to compare how much a given surface behaves like a plane. The gaussian curvature specifically can be described as how much a surface "fails" to be a plain. It can also be roughly approximated by creating geodesic circles around points p on the curved surface with radius r. These geodesic circles have a different total area to a circle created on a plane. The Gaussian curvature can be approximated using the ration of the geodesic circles with center p and radius r with respect to a circle with radius r on a plane. Specifically

$$K \ \propto 1 - \frac{\left|B_g\right|}{\left|B_{\mathbb{R}^2}\right|}$$

$$\left|B_g(p,r)\right| = \left|B_{\mathbb{R}^2}(p,r)\right|\left(1 - \frac{K}{12}r^2 + O(r^3)\right)$$

Where B_g is the area of the geodesic circle around p, K is the gaussian curvature and $B_{\mathbb{R}^2}$ is the area of a circle of radius r calculated on the plane. For very small radii we will say

$$\frac{r^2}{12}K \approx 1 - \frac{\left|B_g(r)\right|}{\left|B_{\mathbb{R}^2}(r)\right|}$$

This means that we can define curvature using only areas without needing to have information about the normals.

1.7.4.4 Gauss-Bonnet Theorem

The Gauss-Bonnet theorem states that the total curvature of a surface is always equal to 2π times the *Euler characteristic* χ.

$$\int_M K dA = 2\pi\chi$$

The Euler characteristic for (close, compact orientable) surfaces of genus g is given by

$$\chi \coloneqq 2 - 2g$$

If the surface has a boundary this value changes to

$$\chi = 2 - 2g - b$$

Where b is the number of boundary components (total boundaries on the surface, for example b=1 for a disc).

For surfaces with a boundary Gauss-Bonnet is expanded to:

$$\int_M KdA + \int_{\partial M} \kappa_g ds = 2\pi\chi$$

This is important as it states that we can have some global information purely by the surface topology.

1.7.5 The first and second fundamental forms

The *first fundamental form* $\boldsymbol{I}(X, \Upsilon)$ is another name for the Riemannian metric

$$I(X, \Upsilon) \coloneqq \langle df(X), df(\Upsilon) \rangle$$

And the *second fundamental form* $\boldsymbol{II}(X, \Upsilon)$**,** is closely related to normal curvature κ_N. The second fundamental form when applied to a pair of tangent vectors X, Υ in the domain of a parametrization is equal to the product of the Weingard map along X with the differential

$$II(X, \Upsilon) \coloneqq \langle dN(X), df(\Upsilon) \rangle$$

The second fundamental form describes the *change* of the first fundamental form under motion in normal direction.

$$\kappa_N(X) = \frac{\langle df(X), dN(X) \rangle}{|df(X)|^2} = \frac{\mathbf{II}(X, X)}{\mathbf{I}(\mathrm{X}, \mathrm{X})}$$

The fundamental theorem of surfaces: *Two surfaces in* $\mathbb{R}^3$ *are identical up to rigid motions if and only if they have the same first and second fundamental forms*. Or in other terms: two surfaces are equivalent if and only if they have the same normal curvature and same Riemannian metric. However, not every pair of bilinear forms **I, II** describe a valid surface. To be a valid surface they must satisfy the *Gauss-Codazzi* equations (integrability conditions).

Fundamental theorem of discrete surfaces: Up to rigid motions, we can recover a discrete surface from its *dihedral angles* and *edge lengths.* Where edge lengths describe the first fundamental form and dihedral angles describe the second fundamental form.

1.7.6 Discrete curvature of surfaces

For discrete curvature there is a unified viewpoint which explains the choices made to adapt different types of curvature from the smooth setting to the discrete setting. We will divide the study of discrete curvature into two sections describing two different viewpoints on how to approach it. An *integral* viewpoint and a *variational* viewpoint.

To simplify notation throughout all this point we will refer to the position of vertices of a trimesh as f_i, e_{ij} is the vector from vertex i to vertex j, ℓ_{ij} is the length of the edge described by e_{ij}, A_{ijk} is the area of the triangle with vertices i, j, k, N_{ijk} is the normal of triangle ijk, θ_i^{jk} is the interior angle at vertex i of the triangle ijk, ϕ_{ij} is the dihedral angle oriented on edge ij

$$\phi_{ij} \coloneqq atan2\left(\hat{e}_{ij} \cdot N_{ijk} \times N_{jil},\ N_{ijk} \cdot N_{jil}\right) \mid \hat{e}_{ij} \coloneqq e_{ij}/\ell_{ij}$$

1.7.6.1 Gaussian curvature

Before defining the gaussian curvature, we first are going to define the Euler characteristic of a polyhedral surface χ.

$$\chi \coloneqq V - E + F$$

Where V is the number of vertices, E is the number of edges and F is the number of faces. χ is a topological invariance of polyhedral surfaces (is independent of tessellation). This is equivalent to the smooth setting definition based on the genus of the shape.

The Euler characteristic is related to curvature because of the ***angle defect***. The angle defect at a vertex i is the deviation of the sum of the interior angles from the Euclidean angle sum of 2π.

$$\Omega_i \coloneqq 2\pi - \sum_{ijk} \theta_i^{jk}$$

The angle defect can be thought of as the integral of Gaussian curvature over a region on a vertex. This can be derived from the same concept we defined in the smooth setting. We can define a circle with a small radius r both on the plane and on the selected vertex p. The area of the plane circle will be $|B_{\mathbb{R}^2}(r)| = \pi r^2$, and the area of the equivalent circle on the surface will be given by the sum of the wedges belonging to each triangle adjacent to i. Each wedge will have an area of

$$W_i(r) = \frac{\theta_i}{2\pi}|B_{\mathbb{R}^2}| = \frac{1}{2}r^2\theta_i$$

The area of the surface circle will be

$$|B_g| = \sum_j W_j\,(r) = \frac{r^2}{2}\sum_j \theta_j$$

Then, by the same principle as in the smooth setting, for small radii

$$\left(\frac{r^2}{12}\right)K = 1 - \frac{1}{2}\sum_j \theta_j$$

Which can equally be written as

$$2\pi - \sum_i \theta_i = \frac{1}{6}\pi r^2 K$$

Similarly, we can use the Gauss map to arrive to the same definition. Given the unit normals on the surface we can map them to a unit sphere, connecting the normals around a vertex on the unit sphere creates a polygon like shape on its surface, and connecting the vertices of that polygon to its center creates a vertex star. The dihedral angles on the original discrete surface are the interior angles of the polygon created on the unit sphere and likewise the initial interior angles become the dihedral angles of the polygon on the sphere. This makes so that the angle defect on the original surface becomes the area of the polygon on the sphere.

This leads us to a theorem about the global structure of our mesh, the *Total angle defect theorem.* The total angle defect theorem says that the total angle defect is a constant, and only depends on the genus (g) of the total surface for its value. For continuous closed surfaces (sphere like) this constant is 4π. This is just an application of the Gauss-Bonnet theorem. To complete the theorem, we define the curvature of a boundary around a vertex as its discrete geodesic curvature

$$\kappa_i \coloneqq \pi - \sum_{ijk} \theta_i^{jk}$$

The total angle defect theorem can be written as: *For a simplicial surface* $K = (V, E, F)$ *with interior angle defects* Ω_i*, and boundary angle defects* k_i

$$\sum_{i \in intV} \Omega_i + \sum_{i \in \partial V} \kappa_i = 2\pi\chi$$

1.7.6.2 Curvature normals

We defined the area normal ($\frac{1}{2} df \wedge df = N dA$), the mean curvature normal ($\frac{1}{2} df \wedge dN = HN dA$) and the Gauss curvature normal ($\frac{1}{2} dN \wedge dN = KN dA$), these are called *mixed areas* of ordinary surface area and area on sphere.

For any surface f with normals N, with principal directions X_1, X_2 we have

Area normal:

$$\begin{aligned} df \wedge df(X_1, X_2) &= df(X_1) \times df(X_2) - df(X_2) \times df(X_1) \\ &= 2df(X_1) \times df(X_2) \\ &= 2NdA(X_1, X_2) \end{aligned}$$

Mean curvature normal:

$$\begin{aligned} df \wedge dN(X_1, X_2) &= df(X_1) \times dN(X_2) - df(X_2) \times dN(X_1) \\ &= \kappa_1 df(X_1) \times df(X_2) - \kappa_2 df(X_2) \times df(X_1) \\ &= (\kappa_1 + \kappa_2) df(X_1) \times df(X_2) \\ &= 2HNdA(X_1, X_2) \end{aligned}$$

Gauss curvature normal:

$$dN \wedge dN(X_1, X_2) = dN(X_1) \times dN(X_2) - dN(X_2) \times dN(X_1)$$
$$= \kappa_1\kappa_2\, df(X_1) \times df(X_2) - \kappa_1\kappa_2\, df(X_2) \times df(X_1)$$
$$= 2\kappa_1\kappa_2\, df(X_1) \times df(X_2)$$
$$= 2KNdA(X_1, X_2)$$

The vector area gave us the average information of the normals on a bounded surface. To obtain this vector area in the discrete setting, we integrate NdA over a dual cell to get the normal at vertex p.

$$\frac{1}{3}\int_{\Omega} NdA = \frac{1}{6}\int_{\partial\Omega} f \times df$$
$$= \frac{1}{6}\sum_{ij\in\partial\Omega}\int_{e_{ij}} f \times df$$
$$= \frac{1}{6}\sum_{ij\in\partial\Omega} f_i \times f_j$$

This provides us with the dual discrete differential 2-form corresponding to the vector area.

The discrete mean curvature can be obtained integrating HN over a circumcenter dual cell C

$$\int_C HNdA = \int_C df \wedge dN = \int_C dN \wedge df$$
$$= \int_C d(N \wedge df) = \int_{\partial C} N \wedge df$$

For a triangulated surface this equal

$$\sum_j \int_{e_{ij}^\star} N \wedge df$$

If we consider the end points of the dual edge $(e_{ij}^\star)$ as a, b and m its midpoint where the dual edge intersects the primal edge, then the previous expression can be rewritten as

$$\sum_j \big(N_a \times (m-a) + N_b \times (b-m)\big)$$

Since both normals N_a, N_b represent the normals on the respective triangles, both resulting terms in the previous expression are parallel to the edge vector e_{ij}. The length of the resulting vector of the sum of the two terms has the same length as the length of the dual edge $(\ell_{ij}^\star)$. We can write the ratio of the dual/ primal length using the cotan formula, meaning

$$(HN)_i \coloneqq \frac{1}{2}\sum_{ij\in E}(cot\alpha_{ij} + cot\beta_{ij})(f_i - f_j)$$

(where again α, β are the angles not touching the edge ij)

Finally, to discretize the **Gauss curvature normal**, similarly as we did to calculate the discrete Gauss curvature, we can project N to the unit sphere but now instead of doing direct connection between the points we use arcs on the unit sphere to connect them.

$$2\int_C KNdA = \int_C dN \wedge dN$$

$$= \int_C d(N \wedge dN)$$

$$= \int_{\partial C} N \wedge dN$$

$$= \int_{\partial C} N \times dN(\gamma')ds$$

Where $dN(\gamma')ds$ is the unit tangent with respect to arc length ($dN(\gamma')$ is the tangent of the curve connecting the projection of two normals on the sphere)

$$= \int_{\partial C} N \times T\, ds$$

$$= \sum_j \int_{\partial C} \frac{e_{ij}}{|e_{ij}|} ds$$

$$= \sum_j \frac{e_{ij}}{\ell_{ij}} \phi_{ij}$$

Replacing e_{ij} with the vertex positions we can write the Gauss curvature normal at vertex i is

$$(KN)_i \coloneqq \frac{1}{2}\sum_{ij\in E}\frac{\phi_{ij}}{\ell_{ij}}(f_j - f_i)$$

1.7.6.3 Steiners formula

Steiners approach is to "smooth out" the discrete surface such that we can compare it to a smooth surface, this will be called a *mollified surface.* And we can precisely calculate the curvature of the mollified surface, the limit as the "smoothing" goes to zero will be the discrete value. This can be done using a *Minkowski sum* (a sum of two sets $A, B\ in\ \mathbb{R}^n,\ A + B \coloneqq \{a + b \mid a \in A, b \in B\}$). The Minkowski sum used to "mollify" a surface consists of adding a sphere with radius $\varepsilon > 0$ to the initial surface. It can be seen as the expanding the surface with a convolution of the sphere through all the surface, smoothing the edges and vertices. This works well for convex polyhedra.

Steiners formula says: Let A be any convex body in $\mathbb{R}^n$, and let B_ε be a ball of radius ε. Then the volume of the Minkowski sum $A + B_\varepsilon$can be expressed as a polynomial in ε:

$$Volume(A + B_\varepsilon) = Volume(A) + \sum_{k=1}^{n} \phi_k(A)\varepsilon^k$$

The coefficients ϕ_k are called the *quermassintegrals (*"cross-dimension integrals"*),* they describe how quickly the volume grows.

The Gaussian curvature K, of the mollified surface with a ball of radius ε can be expressed as a combination of the gaussian curvature provided by the planar faces, which were translated in a normal direction a ε distance, the corners, which became a section of a sphere and the edges, which became a section of a cylinder. The triangles, provide no gaussian curvature ($K = 0$). The edges since they describe a section of a cylinder and cylinders that have null Gaussian curvature for their length, also provide zero Gaussian curvature ($K = 0$). Finally, the vertices provide a polygonal shape projected on top of a spherical surface of radius ε, so their contribution will be $K = 1/\varepsilon^2$ (the product of two principal curvatures with value $1/\varepsilon$). The total curvature provided by vertex i will be given depending on how much area the projected polygon has.

$$A_i K_i = \left(\frac{\Omega_i}{4\pi} 4\pi\varepsilon^2\right)\frac{1}{\varepsilon^2} = \Omega_i$$

Meaning that all vertices have a Gaussian curvature equivalent to their angle defect.

Following the same process we can calculate the mean curvature. The faces still have no curvature ($H = 0$), the edges will have $H = 1/2\varepsilon$ mean curvature, ant the exposed area of the cylindrical piece will be $\ell_{ij}\phi_{ij}\varepsilon$, meaning the total mean curvature per edge will be

$$\frac{1}{2}\ell_{ij}\phi_{ij}$$

Finally, the vertices provide $1/\varepsilon$ curvature in all directions, meaning their total curvature will be

$$\left(\frac{\Omega_i}{4\pi} 4\pi\varepsilon^2\right)\frac{1}{\varepsilon} = \Omega_i\varepsilon$$

Which means, as ε goes to zero, the vertices provide no mean curvature ($H = 0$).

The area of a mollified surface is given by the faces (A_{ijk}), the exposed part of the cylinder for the edges ($\ell_{ij}\phi_{ij}\varepsilon$) and the exposed are of the sphere for the vertices ($\Omega_i\varepsilon^2$).

$$area(\varepsilon) = \sum_{ijk\in F} A_{ijk} + \varepsilon \sum_{ij\in E} \ell_{ij}\phi_{ij} + \varepsilon^2 \sum_{i\in V} \Omega_i$$

The volume of the mollified surface would be

$$Volume(\varepsilon) = V_0 + \varepsilon \sum_{ijk\in F} A_{ijk} + \frac{1}{2}\varepsilon^2 \sum_{ij\in E} \ell_{ij}\phi_{ij} + \frac{1}{3}\varepsilon^3 \sum_{i\in V} \Omega_i$$

Importantly this gives us the following relationships

$$\frac{d}{d\varepsilon} volume_\varepsilon = area_\varepsilon$$

$$\frac{d}{d\varepsilon} area_\varepsilon = 2\, mean_\varepsilon$$

$$\frac{d}{d\varepsilon} mean_\varepsilon = Gauss_\varepsilon$$

$$\frac{d}{d\varepsilon} Gauss_\varepsilon = 0$$

1.7.6.4 Principal curvatures

Given that we know the Gaussian curvature K and the mean curvature H. We can calculate the principal curvatures solving the for κ_1, κ_2 using the formulas

$$K = \kappa_1 \kappa_2$$

$$H = \frac{\kappa_1 + \kappa_2}{2}$$

Meaning

$$\kappa_1 = H - \sqrt{H^2 - K}$$

$$\kappa_2 = H + \sqrt{H^2 - K}$$

In a smooth surface this approach is complete, but in discrete surfaces we defined K as a differential 0-form and H as a differential 1-form. Meaning we need to transform one of them, such that we can calculate the principal curvatures on the desired structures. To transform the mean curvature to a vertex mean curvature we integrate its value in an area around f_i. Using the same Minkowski sum as we did before calculating the Steiner formulas, we calculate the mean curvature over all edges in the dual cell of the vertex. This gives us the expression

$$H_i := \frac{1}{4} \sum_{ij \in E} \ell_{ij} \phi_{ij}$$

Using this discrete area mean curvature, we can calculate the principal curvatures as:

$$\kappa_{1,2} = \frac{H_i}{A_i} \mp \sqrt{\left(\frac{H_i}{A_i}\right)^2 - \frac{K_i}{A_i}}$$

1.8 GEOMETRIC DERIVATION

It is recommended to do geometric derivations thinking about the geometry of the desired parameters to derivate, since classical differentiation can lead to long and unstable solutions. These simpler expressions can be found by asking "what is the direction of quickest change of the studied quantity" and "what is the speed of change of the quantity". This is a process commonly done in geometric computing since classical derivatives tend to be computationally expensive, and for special purpose algorithms this approach tends to give faster and more accurate results.

A second approach use when doing geometric computing is doing numerical differentiation, where we approach the true value iteratively by introducing small perturbations on the initial input and measure the change based on the perturbation value. Is expensive, is less accurate and the perturbation value tends to be hard to select.

1.8.1 Laplace Beltrami operator

The *Laplace Beltrami operator* or *Laplacian* is a generalization of the ordinary Laplacian to curved domains. For simplicity it will be represented as Δ. This is an important operation since reduces many problems to sparse linear algebra problems. In geometry specifically, the Laplacian is used to calculate curvature and frequency decomposition of shapes.

Given a scalar function u: $\mathbb{R}^n \to \mathbb{R}$, which is two times differentiable the classic Laplacian

$$\Delta u = \sum_{i=1}^{n} \frac{\partial^2}{\partial x_i^2} u$$

At maximum and minimum points of functions the Laplacian describes the curvature.

The graph Laplacian L of a graph $G = (V, E)$, where at each vertex i there is a value u_i, gives the deviation of the average of values of all values from vertices neighboring i.

$$(Lu)_i \coloneqq \left(\frac{1}{\deg(i)} \sum_{ij \in E} u_j \right) - u_i$$

Using this same concept, the Laplacian on the surface (u) can be defined as the deviation of values in a sphere around a point x_0.

$$\Delta u(x_0) \propto \lim_{\varepsilon \to 0} \frac{1}{\varepsilon^2} \left(\frac{1}{|S_\varepsilon(x_0)|} \int_{S_\varepsilon(x_0)} u(x)dx - u(x_0) \right)$$

In general, for curved surfaces the Laplace-Beltrami operator can be defined using the Riemannian metric (g).

$$\Delta u = \sum_{i=1}^{n}\sum_{j=1}^{n} \frac{1}{\sqrt{\det(g)}} \frac{\partial}{\partial x_i}\left(\sqrt{\det(g)}\,(g^{-1})_{ij}\frac{\partial}{\partial x_j}u\right)$$

Where n is the working dimension.

Importantly for geometric analysis the Laplacian is invariant to rigid motions and invariant to isometries.

1.9 GEODESICS

A geodesic can be described as the "straightest" curve on a surface between two points. Locally they share the same properties as ordinary "straight" lines (they have no curvature or acceleration, and they minimize the local length). Geodesics can be useful to find the minimal length paths on bounded surfaces, or in curved ones. In domains with boundaries is common for the shortest path not to be straight, in these cases the "straightest" path will at some point follow the boundary in direction towards the end point. Geodesics can be used to correlate figures if the geodesic distance between two points is the same on different surfaces, this property is the *isometry invariance of geodesics*. Isometries are important for mesh analysis as they are defined as special deformations that don't change the intrinsic geometry of a shape. Meaning isometries preserve the Riemannian metric.

1.9.1 Discrete geodesics

Geodesics, like many other quantities, when transformed to the discrete domain tend to be hard to express, as depending on which properties we wish to express a different definition may be better suited. Generally, most definitions don't provide all the characteristics they may have in the smooth setting. First, we will number some of the properties of geodesics and then we will give some definitions which can comply with some of them for simplicial surfaces.

Geodesics are the lines connecting two points on a surface that:

- Are the straightest (have zero acceleration) path between the points.
- Locally are the shortest path.
- Have no geodesic curvature.
- Are a *harmonic map* from an interval to a manifold.
- Are the gradient of the geodesic distance function.

1.9.1.1 Geodesic following the locally "shortest" path

First, we will define that the path of a line is *locally* the shortest for any two "nearby" points on the path, if there doesn't exist a shorter route. We say "nearby" to mean that the shortest path between the two points is unique. Note that a path being locally the shortest doesn't mean it is globally the shortest. For example, if we have two points on a sphere they determine a circle on

the surface of the sphere, this circle will be separated into two segments, generally of different lengths, both these segments are defined as geodesic paths. The shortest of the geodesic paths is called *the minimal geodesic.*

Using the *Dirichlet energy* (an energy measuring "smoothness" or regularity of functions) we can determine the locally shortest path geodesics. Given a planar curve $\gamma: [0,1] \to \mathbb{R}^2$, its Dirichlet energy is

$$E_d(\gamma) = \int_0^1 |\gamma'(t)|^2 dt$$

We can write γ as a reparametrization of a unit-speed curve $\hat{\gamma}: [0, L] \to \mathbb{R}^2$. Using the speed function $c: [0,1] \to \mathbb{R}$, where $c(0) = 0, c(1) = L$, then $\gamma(t) = \hat{\gamma}(c(t))$. Then we know that

$$|\gamma'(t)| = |c'(t)|$$

Minimizing E_d with respect to γ we get the "smoothest curve".

$$\min_{\gamma} E_d(\gamma) = \min_{\hat{\gamma}} \left(\min_{c} \int_0^1 \left(c'(t)\right)^2 dt \right) = \min_{\hat{\gamma}} L^2$$

Where L is the length of the curve. We can find the shortest path minimizing the Dirichlet energy, which can be integrated by parts giving us the following expression:

$$\min_{\gamma} E_d(\gamma) = \min_{\gamma} \int_0^1 |\gamma'(t)|^2 dt = \min_{\gamma} - \int_0^1 \langle \gamma(t), \gamma''(t) \rangle \, dt$$

Taking the gradient of E_d with respect to γ it yields a 1D Poisson equation:

$$\frac{\partial^2}{\partial t^2} \gamma(t) = 0$$

$$\gamma(0) = p$$

$$\gamma(1) = q$$

Where p, q are the fixed endpoints of γ. Importantly since this is given by the inner product this means that this definition works in curved surfaces if we use the Riemannian metric.

$$\min_{\gamma} E_d(\gamma) = \min_{\gamma} - \int_0^1 g(\gamma(t), \gamma''(t)) \, dt$$

On a discrete surface, there are a couple options to find the shortest path on the surface. Intuitively one can think of using an algorithm, such a Dijkstra's algorithm, finding the shortest path from one vertex to another using the connected graph described by the vertices and edges on a mesh.

Dijkstra's finds the shortest connected path, but rarely finds the shortest path on the surface, meaning is almost never a geodesic. A solution can be to locally "straighten" the path found using Dijkstra's algorithm. A solution provided by [3] says that a geodesic path can be created by edge swapping an existing path between two points. Each segment of the path is checked against its swapped edge version, if the distance on the path is shorter with the swap, the mesh is altered to provide the shorter path. A second common solution is to check if the mesh has local discrete curvature (the angle defect) at each vertex in the path, if it has no local curvature the path can pass through the vertex directly. If the vertex is the point of a cone ($\Omega > 0$), that means that the "shortest path" is around the vertex not through it. For saddle vertices ($\Omega < 0$) there are multiple equally valid shortest paths that go through the vertex. Unlike in the smooth setting where geodesics can only intersect if one geodesic is in the path of another, in the discrete setting multiple geodesics can lead to a singular saddle vertex to diverge after it, these points are called *pseudo-sources*.

A *cut locus* is the set of all points q, such that there is not a unique global shortest geodesic from a source point p towards q. The *injectivity radius* is the distance to the closest point on the cut locus. For polyhedral surfaces the injectivity radius forms tree-like structures branching around cone vertices, unlike in the smooth setting where a single line follows the surface.

1.9.2 Geodesics with the "straightest" path

For given a curve we describe it as straight if it has no geodesic curvature or optionally that is has a covariant derivative equal to zero. These two points are equivalent in the smooth setting to saying a curve has no curvature and the curve has no acceleration respectively.

Geometrically, given a curve $\gamma(s)$ with tangent T on a surface with normal N, then $B \coloneqq T \times N$. The bending of the curve is described by its normal curvature $\kappa_n \coloneqq \langle N, \frac{d}{ds}T \rangle$, and its geodesic curvature $\langle B, \frac{d}{ds}T \rangle$. The normal curvature describes the external curvature of γ in the space, while the geodesic curvature describes bending of γ without leaving the containing surface. A discrete curve, for this point, will be defined as a continuous curve γ which is piecewise linear in each simplex of a simplicial surface. These discrete curves are encoded as a sequence of simplices and a set of barycentric coordinates for each simplex. The discrete geodesic curvature at a point connecting two segments is the turning angle κ_i (the externa angle connecting both segments which can be calculated as $\pi - \theta$, where θ is the internal angle between the segments). For points within a face, it's just the external angle between the segments. For a point between two faces (on an edge) the angle still is the same external angle after planarizing the faces. For vertices special care needs to be taken. Using the assumption that the internal and external angle around a point have to have the same value so that that point has no geodesic curvature, then this same definition can be applied to a vertex on a polyhedral surface, where if the sum of the angles on one side of the curve passing through a vertex is the same as the sum on the other side of the curve, we can define that curve to be a geodesic around that the vertex (note that if the vertex has a positive angle defect $\Omega > 0$ both angles will be larger than π).

1.9.3 The Karcher mean

The *Karcher mean* is a definition of the mean value on a surface between several points. It is defined as the point that minimizes the sum of squared geodesic distances from all points.

$$K_{mean} = \min_x \frac{1}{n} \sum_i d(x, y_i)^2$$

The Karcher mean can be calculated using the so-called *log map.*

The log map is the complemental map to the *exponential map*. The exponential map $\exp_p: T_pM$, is the map on a surface M at a point p mapping tangent vectors X to the point on M by walking along a geodesic in the direction of $\frac{X}{|X|}$ for a distance $|X|$. The *logarithmic map* or *log map* can be calculated finding the shortest geodesic γ from p to q, defining X as a vector in direction γ' with length $|\gamma|$. X is called the $\log_p(q)$. Importantly the log map is not always injective, by convention the log map uses the smallest vector X, but if that is not the case, there can be multiple geodesics which provide a correct mapping.

The iterative algorithm to find the Karcher mean is the following: for y_i points on a surface M. Iteratively:

- Pick a random initial point x
- Compute the $\log\ v_i$ at all points y_i
- Compute the mean of v of all v_i
    - If $v = 0$ we are at the mean.
    - Else we move x along v using the exponential map and repeat the procedure until we reach an acceptable threshold.

In general, this converges quickly to a Karcher mean. Importantly the Karcher mean may not be unique (think of the poles of a sphere with respect to the equator).

The *discrete Exponential map* follows the same concept as in the smooth setting but is a bit more precise. Given a point p and a vector u, we can walk along u until we reach an edge. The point reached on the edge becomes a new starting point. Given the angle of the edge with the ray produce by u, we can continue its path on the next triangle setting that the corresponding angle has the same value (Figure 5).

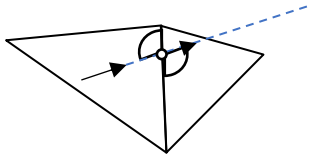


Figure 5 representation of path crossing an edge.

If we hit a vertex, the same definition as the *straightest* geodesic is used. Note the discrete exponential map is not surjective, as saddle vertices have small areas that are not reachable (as these areas overlap when planarizing the faces around a vertex).

2 HUMAN MESHES FOR ANIMATION AND REMESHING

When it comes to 3D modelling, 3D artists generally follow a so-called modelling pipeline, to create the most appealing and functional models possible. This pipeline can be resumed in six major steps: Design & Concept, Sculpting, Retopology, UV Sweep, Bake maps and Texturing. It can later be expanded for animation by adding some extra steps like Rigging and Skinning. This is not a rigid pipeline but are the most common steps recommended when creating 3D models in general, for a more detailed view we recommend [4].

A concept that is suggested to grasp is the part of vertex painting or "Skinning". When Skinning every vertex is given a weight between zero and one corresponding to how much they should follow the transforms applied to a given "bone", note that this weighting normally is applied by distance, and vertices normally have a zero value for all bones which are too distant from them, but can have more than one group with a value larger than zero at any given time. It can be seen as how much each vertex belongs to a certain zone of the mesh.

In our research we focused on the Retopology step or "*retopo"*: Given a 3D high-poly model, generally a trimesh, we create a secondary corresponding mesh, which adheres to some specific constraints. In general, it is required that the new mesh has a close resemblance to the original mesh's shape, and that it is manifold. Two necessary constraints when working for animation are a reduced number of polygons and a correct "flow" of polygons (Figure 6). Specifically, when dealing with human models, it is common practice to prefer a quad mesh, with a specific structure, such that common deformations of the mesh (such as the bending of the elbows or movements of the mouth), are easy to realize and don't cause artifacts when animating.

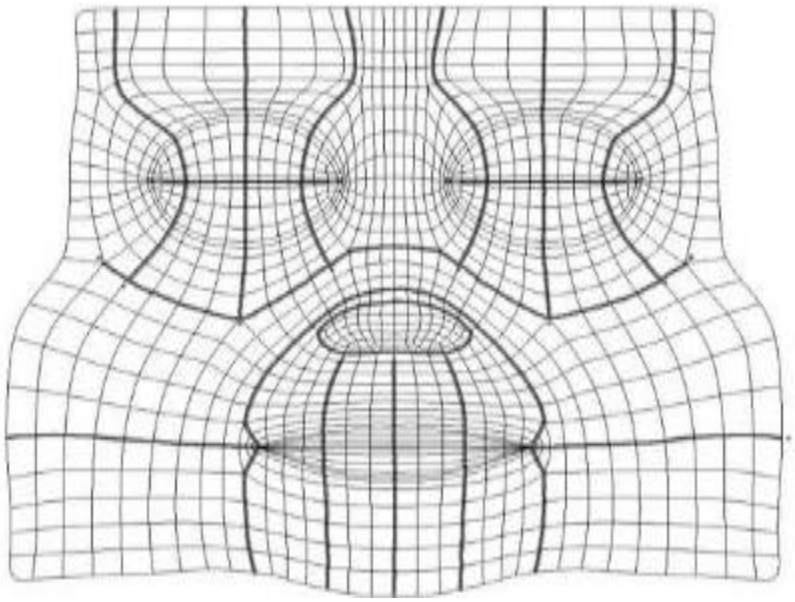


Figure 6 UV representation of the zones describing the geometry of a "good" human face topology [5]

Although not hard defined, after a thorough investigation it was determined that a "*good"* human model, for real time applications, has between 10.000 and 60.000 quads for consoles/PC

applications, for low-end devices the range drops to 1.000 to 10.000 quads [6]. Keep in mind that these are the highest level of detail (LOD) models, in general many applications tend to have multiple models to represent the same character in different LOD.

Shape-wise it must respect some commonly desired topological structures most animators use to have better results. These "structures" we refer are directions the quads should follow on the surface, to allow for smooth deformations when animating. Some of the most important, and more easily observable ones are found on the face, where the flow of polygons follows the natural structures found on the face and the body.

2.1 FROM CONCEPT TO A USABLE MODEL

When creating a 3D model, the three most important steps for a "functional" model are, the concept design, the sculpting and the retopology. If the model is an animatable model, rigging and skinning are essential as well. Other steps like uv-unwrapping and texturing are aesthetic parts of the modelling pipeline and have little to no influence on simulations or real time functionality of meshes but can be used to complement the previous steps making so small details are not lost.

The importance of the design step for real time applications can be considered the definition of the functionality of the mesh. For animated meshes it needs to be defined how they will move, what parts bend and what parts can suffer from deformations and what kind of deformations. If these concepts are well defined, future steps will have an easier time optimizing them. Importantly, the design part also must consider the aesthetic point of view. The projection of the shape of the mesh is done in this step and future steps must try and keep as much as possible the expected shape.

The modelling step is purely an aesthetic step, where the creation of a 3D model representing the shape requested on the previous step is realized. Normally for simple meshes this step tries to optimize the model directly. Creating a model optimizing the number of polygons used. For more complex models this approach is rather hard to implement. The general approach is to create a visually appealing 3D model, and later, on the *retopology* or *retopo* step, optimize it and make it functional. Generally, when sculpting "organic" models is not uncommon to end up with several hundreds of thousand vertices describing a single model. These models are unusable in real time applications as each calculation applied to the model is done to each one of its vertices individually and can lead to long computational times.

To optimize these visually appealing models the general process involves creating a similar enough model capturing the important functional features of the model, and any other details can be later added using normal maps or some other type of textures.

The creation of the second model is the retopology or remeshing step. In general, it is done for all meshes with a high number of polygons and is manually done to meshes which require specific topologies or that will be used regularly. This distinction is important as the manual approach to retopology is much longer and very laborious, but the results are considerably better. For human models it is almost always done manually, but many artists are skilled enough to adapt existing

optimized models to fit the unoptimized mesh, but it is not an easy task and requires knowledge of the correct orientation of different important features to make sure the final result is good.

2.2 FEATURES OF HUMAN MODELS

In general, human models can be approached as different features that need to be interlaced with each other. We call features the different zones or special characteristics of the body that require attention when modeling leading to the insertion of irregular vertices to obtain a specific topology.

These zones can be separated in the head, which is itself separated several times, the arms, the hands, the legs, the feet, the hips and the torso. In each of these zones some feature or features have to be taken into account. For the arms the elbows are particularly important as their bending can lead to artifacts if no additional faces are put in the correct positions and with the correct orientation. We call the addition of extra faces to a base topological form *adding geometry.* The arms and the legs, for the most part, excluding the elbows and knees, are topological cylinders. And the biggest feature that requires attention when doing the retopology of an original mesh is the alignment of the connecting edge loops, meaning the vertical lines up and down the arm connecting them to the hands and the shoulders.

These feature zones are modelled in such a way that the connection with other feature zones can be done easily by adjusting the number of edge loops perpendicular to the boundary edge defining them. The usual requirement to do this is that the boundary is topologically xy-symmetrical with respect to its plane. Meaning that the boundary has both in the horizontally and vertically direction the same amount of vertices with perpendicular edges to the boundary, if the boundary was divided in 4 quadrants each of them should be a combination of flips with respect to each other, only taking into account the connectivity of the vertices, in this situation to make sure the flips are equivalent the first ring of faces is considered the judging ground for the boundary instead of the edges.

2.2.1 Hands and feet

Hands similarly to the arms have zones (joints) that require extra geometry to allow for bending. The zones to take special care of on the hands are the finger knuckles and the finger connections, with a special mention to the thumb as it requires special treatment. From a topological point of view each finger can be seen as a cylinder with two “elbows”, rigid zones such as the fingertips in general don’t require a specific topology can be left to the criteria of the artist on how to handle them. A common approach is to have four irregular vertices on the “corners” with a positive angle defect allowing for a topology similar to that of the cube.

For the finger connections in the hand a knuckle must be created to allow bending. And for the thumb a general approach is to segmentate a zone in the palm, create a separate region to simulate the first thumb segment and rotate it 90-degrees, after which the thumb can be treated as any other finger.

The connection of the hand with the arm is done by making sure the hand has a symmetrical edge, and in case extra edge loops are required, it is more common to add them to the arm than to the hand.

For the feet the toes are treated similarly as the fingers are in the hand but don't require knuckles for bending on the connecting segment to the foot. Besides that, irregular vertices are added to consider the bending of the heel.

2.2.2 Torso and hips

The torso and the hips must take care with the connecting segments to the other parts as if these connections require a high range of motion a lack of care can lead to them looking fake or uncanny when moved. The main feature that requires preservation are the shoulders (normally done by a double extrusion from half the connecting ring of the torso), the bust (specially for female models as the chest requires extra topology to allow for extrusion without an outrageous amount of topology to the full torso), and the buttocks (similarly as the chest can require similar extra care).

2.2.3 Head and neck

The neck is a pure cylinder, and the creation of any irregular vertices is done in the head, which generally is the most delicate part of the whole process.

The head can be modelled following the features of the face and later complementing them. Similarly, as previously said, the best approach is to segmentate the faces into different feature zones and afterwards connect them carefully.

The main zones of the face are the brow, the eye sockets, the nose, the mouth and the ears. Besides these zones we will just mention the jaws, the cheeks and the back/top of the head, all of which can have irregular vertices but that can be something each 3D artist decides depending on the objective it has for the mesh.

Most of the irregular vertices on the face are found in the connection of these main zones. Each of these zones have a specific topological shape based on the type of movements they are predicted to have. Zones such as the mouth and eye lids have pivot-points which allow them to emulate the motion of opening and closing. The separation of some of these zones, such as the brow, allows for their deformation making sure that only the next connected zone is affected. In general, a good face mesh will have a uv-map like in Figure 6 UV representation of the zones describing the geometry of a "good" human face topology And the connection to the rest of the head can be thought as the connection of a square patch on a uv-sphere.

2.3 SKINNING

The existence of these guidelines for artists justifies the segmentation of meshes into feature vectors representing their "zones". Specifically, in 3d modelling a technique called skinning does a similar thing. Given a skeleton belonging to a mesh (used for rigging and is only related to the anatomical skeleton in the sense it must imitate its functionality, but in general is not close to its shape), to each vertex of the mesh a vector of weights will be assigned to it. These weights determine the influence each "bone" of the skeleton will have on the deformation of the vertex. Where a value of 0 means the bone will have no influence on the vertex and a value of one means the vertex will suffer the exact same deformations and rigid motions as the bone. The final position, rotation and scale [2] of each vertex is determined by a weighted sum of the influence of each bone on it.

This step in the animation pipeline adapts itself neatly to machine learning, as a good way to predict features on a mesh can be directly represented by weights bones have on each vertex.

Just as when mapping parametrizations to a surface the value on any point between vertices can be done as a weighted sum of the corresponding barycentric coordinates when dealing with trimeshes.

[2] Scale in this case means the distance from the vertex to the corresponding bone pivot, where a smaller scale makes the vertex get closer to the bone.

3 REMESHING

We have talked about the need for a restructuring of an existing mesh, this process, in general, is called **remeshing**. Multiple standard techniques are used to drive algorithms to create meshes with more desirable properties. These properties include but are not limited to regular polygons (triangles in general), with both regular sides and internal angles, vertex regularity (normally aiming to have 6 triangles emerging from each vertex, for trimeshes or 4 quads for quad meshes), removal of short edges and general defects, reduction of total polygons. Furthermore, the

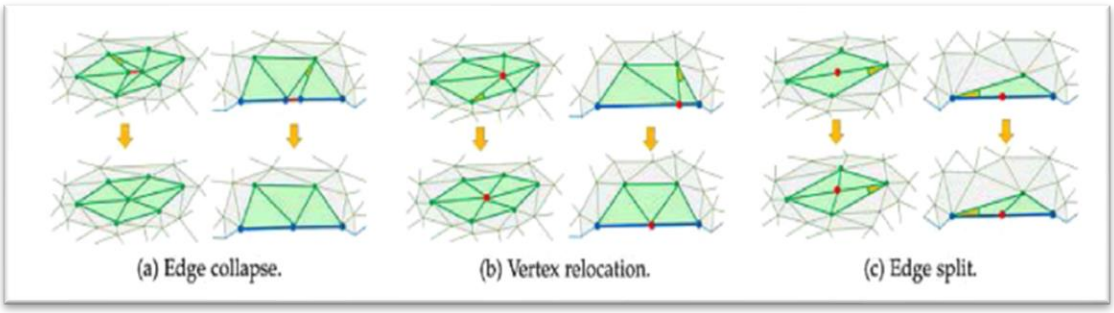


Figure 7 Local operators, where top shows regions before the application of a local operator and the bottom shows the results. With each figure showing both inner regions and boundary regions to the left and right respectively [10] (Copyright 2017, IEEE).

remeshing algorithms should consider existing problems known in the field such as conservation of sharp features, mesh validity (ensure that the resulting mesh is closed and simple manifold) and evaluate their performance both in time and used resources. Note that most algorithms decide to focus on a few of these properties at a time, having worse performance when evaluating others but having better results for their own objectives.

We will consider retopo as regular remeshing for the purposes of this thesis, but we want to clarify that in general retopo is remeshing done manually by an artist, with the objective of optimizing their mesh for either animation or real time applications. When doing retopo the artist follows the shape and curvature of their mesh to improve the mesh's topology. It has better results than automatic remeshing but is very laborious.

The literature review was done approaching the problem step by step. Meaning, first we dealt with an investigation on remeshing in general, which was primarily focused on general concepts and their applications, and afterward we focused our efforts investigating "quad-meshing" and intelligent remeshing [3].

3.1 REMESHING TECHNIQUES AND ALGORITHMS

Following [7], as a guide, we investigated the state-of-the-art for remeshing. When approaching remeshing, rather than specific algorithms, a combination of them is applied. We will broadly

[3] Intelligent remeshing: refers to all remeshing done using any kind of ai tool.

describe some categories which describe most of the current approaches, considering that most algorithms mentioned will take part, in general, to more than one category.

3.1.1 Simplification:

Simplification is the reduction of the level of detail of a mesh to make them more efficient for applications, using different methods trying to balance efficiency and memory usage.

A regular technique for simplification is the *half edge collapsing scheme*, which consists in collapsing half-edges (edges) without the creation of new vertices [8].

Half edge collapsing has been applied in an algorithm called *Normal Field deviation:* using a greedy algorithm, calculate locally the normal field of the ring around a vertex, collapsing some of the corresponding half edges causing minimum disruption to the local shape (according to a given evaluation function) [9]. If the collapse of any of the edges is considered too large a disruption the vertex is left as is.

3.1.2 *Local modification*:

Local modification consist on the transformation of the mesh using local operators such as: edge flipping, edge collapsing, edge splitting, vertex translation [10] Figure 7.

In general, these operations aim to modify the mesh quality rather than reduce the poly count it has.

Edge flipping can be described as the selection of the second diagonal of a quad represented by two adjacent triangles. E.g. when given 4 vertices (A, B, C, D) belonging to two triangles, ABC and DCB for example, the flipping of the common edge, is the transformation from the initial triangles to two new triangles with a different common edge but belonging to the same initial quad. In our example from ABC and DCB we would create ADB and DAC.

Edge collapsing refers to when: an edge has a length $\leq \varepsilon$, it is selected for collapse meaning that the vertices that form it are moved to a single position, generally to either their mid-point or to the initial position of one of them. When doing so, special attention must be given to eliminate and modify correctly the adjacent facets from all the corresponding data structures. This scheme collapses a quad formed by two triangles, into a line formed by two edges, normally these two segmented edges are further simplified into a single edge.

Vertex relocation is simply the change of coordinates of a specific vertex, normally it is done carefully to not disrupt the working surface, and the objective is generally to improve triangle quality. This relocation does not change the topology of the mesh.

Edge splitting consists of selecting a point on an edge, dividing it into two sections, after which this new vertex can connected to the two closest vertices which do not form an edge with it already or alternatively can be used to create a new quad formed by two triangles (following the opposite procedure as in edge collapsing).

Notable in this category is the use of real-time adaptive remeshing (RAR) [9], which is an efficient way of using local operators. Some software that use RAR include Blender and 3D-Maya.

3.1.3 Segmentation:

When working with local operators a per region process is realized, this however is not a separation of the mesh and the whole figure must be considered when realizing error acceptance functions. Meanwhile Segmentation works by subdividing the original mesh into separate meshes and working them independently, after which they are "stitched" back together.

When talking about segmentation two approaches are considered the norm: Defining a coarse/base mesh over the input mesh, commonly using a method such as simplification, after which this base mesh is remapped back onto the input mesh and further developed in order to have a more structured output, this category is commonly used for the definition of feature skeletons such as exoskeletons and LiveWire [11]. A second option is the creation of patches which are adapted to the geometry of the mesh independently and are expanded until they collide, after which they are stitched together creating a new mesh. [12] is a proposed method using this approach to do surface remeshing taking particular care on the preservation of sharp features.

3.1.4 Delaunay triangulation (DT):

The Delaunay criterion [13], states that the circumscribed circle of a triangle will not have nodes inside of it (Figure 8). DT is used to try and improve triangle quality on the working surface, by both deleting and adding vertices and edges as the algorithms see fit. Optimal DT (ODT) [14],

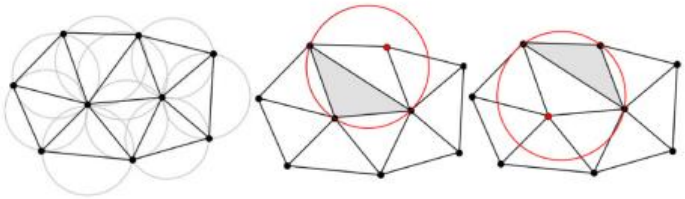


Figure 8 Delaunay criterion. On the left there's an example of a mesh where the circumcircle of each triangle does not contain any node. The middle and the right side violate the Delaunay criterion [65].

uses both local and global smoothing interpolation schemes based on minimizing an error interpolation function, allowing for fast optimization based on a greedy optimization using the Delaunay criterion.

3.1.5 Advancing front:

The advancing-front method is based on the concept of sequentially creating a mesh on an element-by-element basis.

One classic example of this is the marching cubes algorithm (MC) [12], where given a simple mesh a cube can be transported on its surface following a straight path in all directions until it collides with its own path. It can be visualized as a cube leaving a path of quads on the surface of the mesh. The final resolution of the mesh is given by the resolution given by the advancing cube, the smaller the cube the higher the resolution. An adaptable resolution is somewhat possible but is an algorithm being actively researched. Due to the regularity imposed by the cube measurements, the creation of poor elements is common or this algorithm.

MC was improved with the *MC using edge transformation*, which can solve some of the problems presented by MC, such as the creation of some poor elements. Although it provides better results than MC, it is not always applicable to all meshes that can use MC, such as those with sharp corners or adaptive versions. Algorithms that tackle these issues are stated as a future research direction.

A problem commonly stated when dealing with advancing front methods is their low efficiency, especially when dealing with highly complex meshes. Furthermore, ensuring that the results are not self-intersecting is an issue as well, as it takes considerable time and effort and further reduces the efficiency of the algorithm.

3.1.6 Laplacian smoothing:

Laplacian smoothing [15] is considered the simplest method for mesh smoothing, lowering the variance of triangles. The simplest implementation is the change of position of a vertex to the average position of all its neighbors (Figure 9). Although attractive due to its simplicity, it is not guaranteed to converge, often showing beneficial results only at the first steps. Furthermore, it has the risk of changing the direction of faces affected by the vertex movement, and a common negative consequence of this operation is the shrinkage of the mesh and loss of detail if special care is not taken.

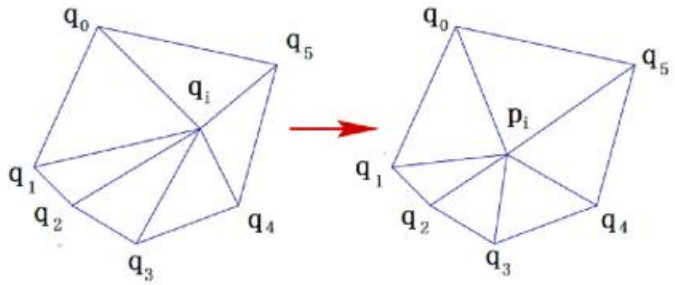


Figure 9 Example of Laplacian smoothing [16]

Some of these issues are addressed by weighted Laplacian smoothing [17], which improves mesh quality and feature preservation, however the weights and features must be set manually.

3.1.7 Optimization:

Optimization based methods are subdivided into two categories: Local operation optimization and global energy optimization. Most of the methods described previously can be used in local operation optimization. Global methods can further be sub-categorized into parametrization-based methods, discrete clustering, and direct 3D optimization. These three methods are normally used as a regularization step on the mesh such that following steps become simpler by working on global structures rather than on a face by face basis.

3.1.8 Centroidal Voronoi Tessellation (CVT):

The Voronoi diagram (Voronoi tessellation) is closely related to the Delaunay triangulation, defined as the dual of it. A Voronoi tessellation is defined central when the generating point is the centre of mass or mean of the corresponding generating vertices. Commonly the Voronoi diagram is generated by minimizing an energy function based on a seed, for CVT of *n* distinct vertices. The energy function is defined as:

$$F_{CVT}(V) = \sum_{i=1}^{n} \int_{\Omega} \rho(v) \|v - v_i\|^2 \, d\sigma$$

Where $\Omega_i = \{ v \in R^3 \mid \|v - v_i\| \leq \|v - v_j\|, \forall j \neq i \}$ is the Voronoi cell corresponding to vertex v_i, and $\rho(v) \geq 0$ is a density function defined over the domain.

Two popular implementations are known as the Lloyd method [18] and quasi-Newton-like solver optimization [19].

The Lloyd method [18] is simple, moving each vertex to the corresponding centroid of the generated Voronoi cell.

We used an existing implementation of CVT loosely adapting [20, 21, 22] for data augmentation, this method is based on Centroidal Voronoi diagram on calculated on the surface of a trimesh as a starter, using a clustering algorithm to normalize the distribution of the cells on the surface and afterwards create a mesh triangulating the clusters using the dual of each cluster to triangulate the new mesh.

3.1.9 Anisotropic remeshing:

Unlike isotropic remeshing, which focuses on regularity of face shape throughout the whole mesh, anisotropic remeshing is adaptive to the local surface [23]. Normally, anisotropic meshes must be evaluated using “anisotropic metrics” (unlike the isotropic Euclidean metrics used in the previews methods), such as Riemannian metrics (specified by the user or surface curvature tensors). An example of an operation used to judge anisotropic meshes is the use of geodesics to create a correspondence map.

3.2 QUAD-MESHING

For our purposes, general meshing was a good guide and provided useful tools, but it revealed itself to be incompatible with creation of specific topologies and its methods are not always generalizable to quad-meshing. Given our focus on human meshes for animation, we decided to investigate quad-mesh generation. Although similar in premise to general remeshing, it presents some exclusive constraints and properties which lead to unique solutions, not compatible with general remeshing algorithms.

To estimate the shape approximation quality a common measurement is Haussdorf distance, efficiently calculated by [24]. For our purposes rendering methods such as silhouette comparison and equal angle rendering comparison are of similar value, as shown by [25]. These last two methods judge the similarity of different meshes based on the global geometry they have based on their total representation but do not have input about nether the local geometry nor the topology of the mesh.

When talking about shape quality evaluation some new problems arise for quad meshes. Quads, unlike triangles, do not necessarily have to be flat in a plane. This becomes a problem since non-convex quads are considered problematic for most applications.

Ideally quads should be "triangle like", in the sense that are flat on a plane (property often categorized as planar-quad, PQ). In most applications, it is desirable that the internal angles of the quad are as close as possible to $\frac{\pi}{2}$, and opposite edges of a quad should have the same length (this is particularly important for simulations).

As for the orientation and sizes of the quads, it depends on the application and the algorithm used. Commonly, an approximation of the principal curvature of the surface is used as a guide, or in other cases principal features such as Langer lines can be used as guides [26] (more details and examples will be shown 3.2.1.1).

In some cases, resolution adaptivity is desirable, allowing the tessellation density to vary on a surface is a great way to reduce the number of surface polygons, allowing only the more "detailed" or "active" sections of a mesh to have a high concentration of polygons. This, however, comes with a tradeoff since the change of resolution forcefully introduces irregular vertices.

The existence of irregular vertices is related to another property that must be studied, which is the connectivity and regularity of the mesh. Meshes are categorized as "unstructured", "valence semi-regular", "semi-regular" or "regular" depending on the regularities they present. The desired category is application dependent. Irregular vertices tend to present problems for most applications, such as wrinkling or curvature irregularities, but this is a necessary sacrifice since depending on the surface (and its shape genus), a set of irregular vertices is necessary to represent it. A normal solution is to have irregular vertices planned in strategic places, such that their interference is rendered to a minimum.

3.2.1 Quad mesh generation and remeshing

The creation of quad meshes, like general remeshing, can be categorized depending on the objectives for the meshing, and the methods used.

A common approach is to transform triangular meshes directly into quad meshes. A naïve application of this practice is the use of Catmull-Clark subdivision [27], which converts every N-gon face to N different quads. It has the drawback that the polycount increases rapidly.

A second approach for quad mesh generation is a patch-based approach, which works by creating a 1-to-1 mapping from sections of the mesh to a set of square patches, which will guide a simpler remesh and can later be "stitched" together. The remeshing of each patch becomes trivial in concept using subdivision or sampling on a regular grid for each patch, but the resulting mesh will be semi-regular by construction.

Thirdly, there exist parameterization-based methods. Assuming that the surface can be cut by a set of curves (seams) into one or more topological disks. The main principle of parametrization-based methods consists of mapping a working surface embedded in 3D space to a domain in 2D, such that quadrangulation becomes trivial. Usually, the new domain is tessellated by regular tiling, such as in the case of a Cartesian grid formed by integer isolines (the regular cartesian plane). In differential geometric terms, we map it to a regular constant 2-form.

Finally, we have field-guided methods, these methods are characterized by controlling the desired local properties of a mesh using a guiding field. Typically, algorithms work with 2^2-RoSy fields, specifically the cross field matching the principal curvature lines of the surface. Using the principal curvature is the most common approach since the calculation of principal curvature can be done automatically. Direction fields defined by 2^2-RoSy fields present the singularities equivalent to irregular vertices present in quad meshes. Many methods use either manual or automatic fields as their input. In some cases, two complementary fields are calculated, one field designated to guide orientation and one field to guide sizing. In the case of frame fields, the direction field and orientation field are combined into one.

Field guided methods consist of two major steps: **field generation** and **mesh synthesis**. Reducing the complex task of quad mesh generation into two relatively simpler subproblems. As stated by [28], this alone already makes them an attractive alternative, since both parts can be optimized

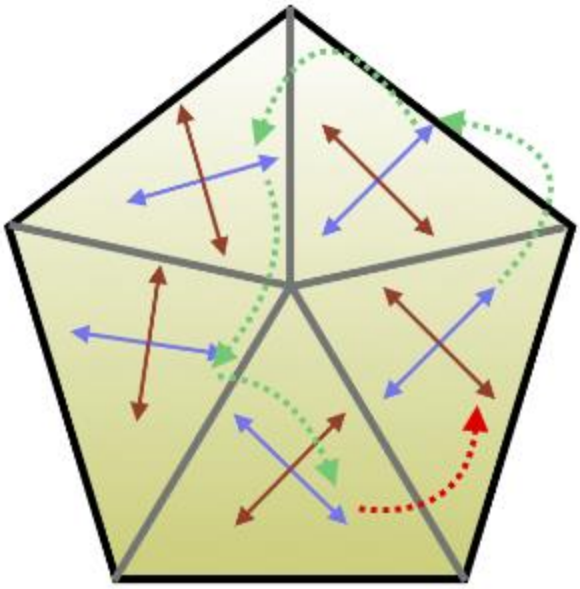


Figure 10 [79] Example of a singular cycle around a vertex

separately, which is much more palatable than the optimization of the whole problem.

3.2.1.1 Field-oriented remeshing: field generation

When dealing with field generation, a new subdivision can be effectuated, dividing the field generation into orientation field generation and sizing field generation. For our purposes we consider orientation as a slightly more relevant parameter to judge but when proposing our model we will try predicting both.

We work with *frame fields* [29], derived from the work of the called cross-fields [30] (further details and a in depth explanation will be given later in 3.3). It's important to highlight that multidimensional orientation fields behave differently from direction fields, i.e., single unit vector fields. Which eliminates some approaches used for them, such as [31] and [32].

A frame field can be described as a four interlinked direction fields d_i, where i is an integer number between 0 and 3, and pairwise d_0, d_1 are antisymmetric to d_2 and d_3 respectively. However, it is to note that the space explored by the frame field is richer than that of a linear combination of four independent direction fields. To exemplify this point, when surrounding a singularity, these direction fields can rotate around it in a small loop, such that after a full cycle the corresponding cross has rotated a multiple of $\frac{\pi}{2}$ (as shown in Figure 10), there showing that the jump from one vector field to another is possible.

Methods creating fields can rely on an optimization problem to create the desired fields. [33] relies on minimizing a quadratic smoothness energy defined over the surface, configurable by the positioning of the desired singularities and their desired degrees. As a secondary benefit it was designed so the method can closely follow guiding direction lines not belonging to the original field. The optimization of the smoothness energy is done by iteratively optimizing the gradient of the energy on the surface, doing small steps in the opposite direction of the gradient.

The method we follow to project our model is based on [34] work, where they trained a neural network to predict on a per face basis a frame field based upon a dual architecture, using PointNet [35] and SpiralNet [36] to extract local and global features, using an MLP as a convolutional operator for the networks they extracted a predicted frame field per each facet.

3.2.1.2 Field-oriented remeshing: mesh synthesis

Once a guiding field is defined, most algorithms work on synthesizing a new mesh following the given field and the initial structure of the input mesh. Once again, we find two mayor categories which describe most algorithms: explicit methods and global parametrization methods. The first method consists of generating curves which follow, as closely as possible, the guiding field over the given surface. Global parametrization, on the other hand, searches for a mapping in a function space.

A 2D application of these concept [37] uses computer vision to align satellite images with a generated cross field following the edges of buildings, after which these fields guide the curves identifying the segmentation of the buildings.

[38] creates a quad layout without the need to create a global parametrization, by tracing geodesic paths following the separatrices generated by a field, generating a T-mesh and the corresponding graph G = (V, E), the optimization of G allows for the creation of simple patches for remeshing. The optimization of G consists of the cancellation of 0 valued parametrized edges and optimizing the number of connections needed to represent a shape.

3.3 STUDY OF A FRAME FIELD

To make sure that our process manages the results we get in an efficient manner, we study frame field and directional field processing.

3.3.1 Directional fields

A definition of a directional field defined by [31] describes a tangent N-directional field as a map

$$V: T\Omega \rightarrow \mathbb{R}^{Nx3}$$

Where $T\Omega$ is a tangent bundle and N vectors are described by (*x, y, z*) coordinates. The tangent bundle is defined by *N*x2 *degrees of freedom (d.o.f.*) at each point p where:

$$V(p) = \{v_i(p) | \forall i, v_i \perp N(p)\}$$

In general, the tangent bundle is described by the set of orthonormal bases $B_1(p), B_2(p)$. And the directional fields are described in cartesian coordinates *(u, v)* following these orthonormal basis.

For our purposes we work with frame fields, defined specifically as a 2^2-RoSy field described by 2 vectors < U, V>, where there is an angle between 0 and π between V and U. Each regular point on the surface is described by the ordered set of vectors <U, V, -U, -V>.

For the sake of brevity, we will only discuss face-based representation, meaning that for each face the barycenter of it will be considered as the point p representing it. Conveniently, faces are tangent planes, which leads neatly to the creation of our desired basis. But we are compelled to mention that both an edge base representation and a vertex based one exist and have been used in the past, but have shown to be less stable, being more sensitive to vertex position changes.

3.3.2 Discretization

We are working in a discrete field; this provides several problems that do not exist in the continuous setting.

- Connection and parallel transport
- Interpolation
- Matching
- Sampling

Discrete connections and parallel transport create a problem due to the difference existing between the tangent spaces. The most common solution implies transforming one vector from one base to another. It can be described as flattening both bases $B(i)$ and $B(j)$ such that they both are in a single plane. After which a direct transformation from the first base to the second becomes trivial. This method is described as flattening plus single axis system. It is necessary to specify when dealing with more than one vector per field to which corresponding vector they are mapped when parallel transported along the surface. In general, it is not defined to which vector from j corresponds to each vector in i, therefore adding the need to deal with matching.

3.3.2.1 Matching

First it is needed to determine if it's necessary to conserver order, if there are N vectors to match this reduces the space of possibilities to N. After this it is the most common approach is to reduce the "*effort*" needed to translate from i to j, for this we select the so called *"minimum effort angle"* or "*closest angle*".

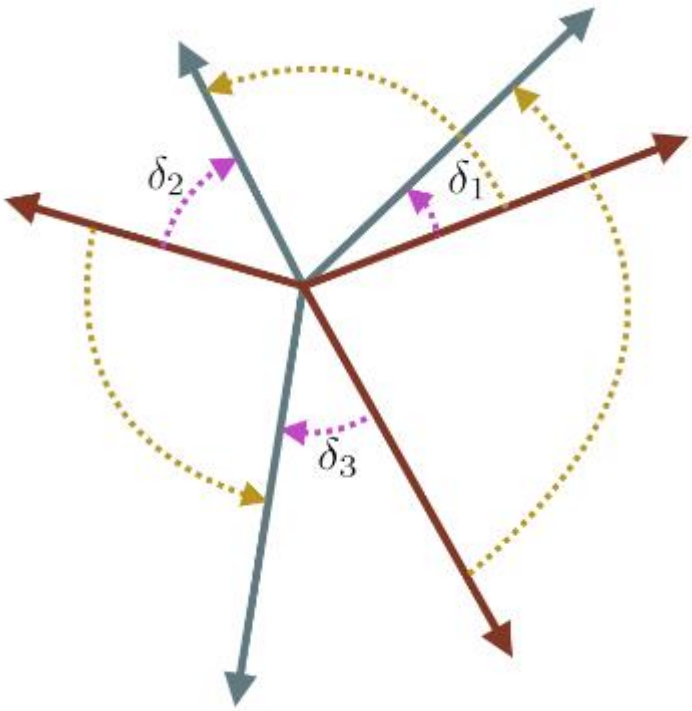


Figure 11 Example from [39] "minimum effort angle"

As shown in Figure 11 Example from "minimum effort angle" there are N possible rotations to be considered. But if we do a signed sum of all the rotations which maintain order separately, we end up that the minimum effort energy $Y = \Sigma\delta$ and other effort energies $Y_\tau = Y + 2k\pi$. Defined $Y \in [-\pi, \pi)$ as the *principal matching*.

When matching around a vertex there exists the possibility that the "return vector", does not match its original position on its original face. Meaning that if we do matching to all the faces around a single vertex, in order, from an initial face f_i to its subsequent faces f_{i+1}, f_{i+2},…f_j, keeping the position from the last checked face, when it gets to face f_j and we match it one final time back to f_i we may have a mismatch of the position of the transferred vector (a visualization can be seen in Figure 10). In this case the cycle is called singular, and the number of "*clicks*" it was translated to the left or right is the index of the singularity ($\frac{1}{k}$) and it has an induced curvature of $\frac{2\pi}{k}$. Following the Poincare-Hopf theorem it is necessary that the sum of all the singularity indices equal $2 - 2g$ where g is the genus of the total surface. We can study these singularity points similarly as umbilics when we reviewed main curvature directions. Notice that the main curvature directions are defined as a cross-direction field and the main curvatures are the corresponding sizing field.

The process in which on a mesh we reorder the vectors <U, V>, if necessary, to match those in the adjacent face is called combing, it is regularly done when generating algorithms as combed fields tend to be easier to analyze.

3.3.2.2 Interpolation

Given a matching pair of vectors V_i, V_j on a pair of parallel planes, the interpolation described between the two can be described as:

$$\delta_{ij} = \Theta + 2\pi k, k \in \mathbb{Z}$$

Where Θ is the principal rotation and $2\pi k$ describes possible period jumps that can happen between V_i and V_j. The principal direction is defined as the shortest rotation needed to transform V_i to V_j there implying $\Theta \in (-\pi, \pi]$, the existence of these period jumps may be described explicitly if we use a polar representation and, in some cases, they are created by the nature of the field itself.

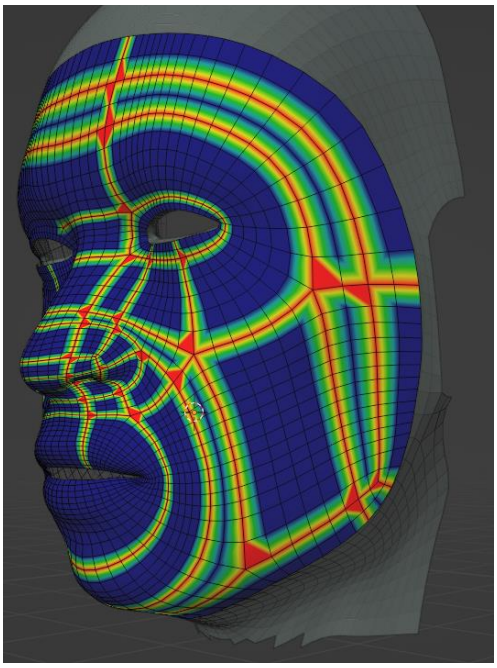


Figure 12 face mesh showing separatrices

3.3.2.3 Sampling

When dealing with discrete data, sampling can create a problem in general. When working with directional fields this can lead to so called singularity parties, which are groups of singularities which on total can ether cancel one another or show the existence of other opposite "parties" on the mesh, or even "mask" a larger singularity that should take their place. This can lead to problems with future algorithms that use the field, for example when tracing the separatrices, if instead of one singularity we have a party representing it we would have a large amount of separatrices that slow down our calculations and may lead erroneous solutions. A common solution is after calculating the field singularities, they are examined with other spatially closed singularities and if possible, they are reduced to one which has a specified border used to calculate related quantities (such as separatrices) and has a specific degree. In some cases, this grouping leads to the cancellation of complementing singularities.

3.3.3 Separatrix directions

Singularities, for a face defined direction 2^2-RoSy field, are defined in vertices (singular vertices). The corresponding separatrix directions to each singularity can be calculated following a simple procedure:

For all faces around a singular vertex compare the combed frame field in two adjacent faces with the direction vector connecting each face with the singular vertex v_i, where i is the index identifying the face and the combed direction field uses the vectors best matching the vertex direction to do calculations. The rotation of both fields with respect to their vertex vectors v_i, v_j are opposite in direction, it means there is a separatrix in between them. The separatrix can be calculated as the linear interpolation of the field vectors which best match the vertex vector

weighted by the mismatch angle each of them has with the corresponding direction vector v_i, v_j. To make sure there are no angles close to π, each face can be evaluated as an equivalent trio of faces, where each new face is described by the angle at the singular vertex divided by three, and a third of the edge opposite to said vertex.

4 MACHINE LEARNING

Machine learning is a field of study designed to allow statistical algorithms to "learn" from data to realize automatic predictions. An "agent" learns when, given "experience", it improves their performance. Machine learning is the association of a set of inputs (X) to a response by a system to them (Y). Based on the nature of both X and Y and the presence of Y apriori, we can observe four kinds of scenarios.

Supervised learning:

These models are realized by giving the algorithm a set of data with the corresponding output values. The training data consists of training examples. Each training example is a set of inputs X and a corresponding correct set of outputs Y. The final objective of these models is to learn a map: $h : X \rightarrow Y$.

Unsupervised learning:

The algorithm has no information about the output categories, and the system can only improve through feedback on the correctness of its response. The final goal of the algorithm is to learn to recognize patterns on the initial inputs to classify them.

Semi-supervised learning:

These types of algorithms use both labeled and unlabeled data to realize their "learning" in a more adequate manner.

Reinforcement learning:

The program interacts with a dynamical system, and the objective of it is to maximize a reward function. Given that the inputs of the program are, normally, affected by their dynamical system, the program can only improve via trial and error. In other words, X is defined by previous states of the system.

4.1 SUPERVISED LEARNING

Supervised learning is a form of machine learning which uses labeled data sets to train algorithms, making them induce a map from the initial data to its labels. This mapping function can be used to identify patterns in the initial data to be used to refine predictive models or inform decisions during automated workflows.

Supervised learning has the advantage of working with known quantities, such as features or expected outcomes. In most cases these quantities are **Ground Truths,** which is information defined as "real", correct or true, provided by empirical evidence (normally direct observation and measurement). Working with these ground-truths can speed up the review process, as a standard metric allows the developers to judge the effectiveness of their models more easily.

The combination of input data and the corresponding response to it is known as *Training set.* Each training data point is represented by a vector known as *feature vector,* which contains all the quantities useful to describe each champion. The total training data is represented by a matrix, where each row represents one data point, and the columns represent its features.

The objective of supervised learning is to create a model, that given an input which wasn't part of the training dataset, to correctly predict its output. This is done via the iterative adjustment of the parameters in the network, using an objective function as a guide.

We can formalize datasets as:

$$D = \{(X_1, Y_1), \dots, (X_n, Y_n)\} \subseteq \mathbb{R}^d \times \mathcal{C}$$

Where D is the entire dataset, n is the size of our dataset, $\mathbb{R}^d$ is the d-dimensional feature space, X_i is the feature vector of the i^{th} example, Y_i is the corresponding label or output of the i^{th} example, and $\mathcal{C}$ is the space of all possible labels, or label space (the label space can be a continuous range with well-defined limits).

We describe the goal of supervised machine learning as finding the map $h : \mathbb{R}^d \to \mathcal{C}$ such that for every new input/output pair (x, y) we have $f(x) \approx y$.

Supervised learning can be studied in different forms depending on the support of Y. If Y is discrete is regularly a classification problem, while if its contiguous is a *regression* one. A typical example of a classification problem is differentiation of images into categories or detection of spam emails. In these cases, $\mathcal{C}$ either is a range $\{1,2, \dots, K\}$ for a multi-class classification problem as in the image classification problem or a reduced space like $\{0,1\}$ or $\{-1,1\}$ for binary classification, as in the spam detection problem. An example of regression can be temperature prediction of a certain area given its history and recent weather patterns, or height prediction of individuals given their family history.

The development of a supervised model must start by defining an exact goal, be it translation of handwriting or image segmentation, clearly defining the problem statement is crucial to be able to evaluate correctly the predictions given by the model. Once the problem statement is well defined it is possible to express what characteristics a suitable dataset must have to provide a good evaluation. The dataset is one of the most crucial factors when training a model, low-quality datapoints may "poison" our model leading to worse results due to their low-quality erroneous outputs, or even just corrupted initial values.

Before designing a model, it is important to estimate the shape h should have. Be it linear functions, decision trees, polynomials or others. These are hypothesis spaces, denoted as $\mathcal{H}$. This initial assumption will impact how the designed model will predict unknown data. If the selected space is not selected carefully it can lead to overfitting or underfitting depending on the situation, as seen on Figure 13.

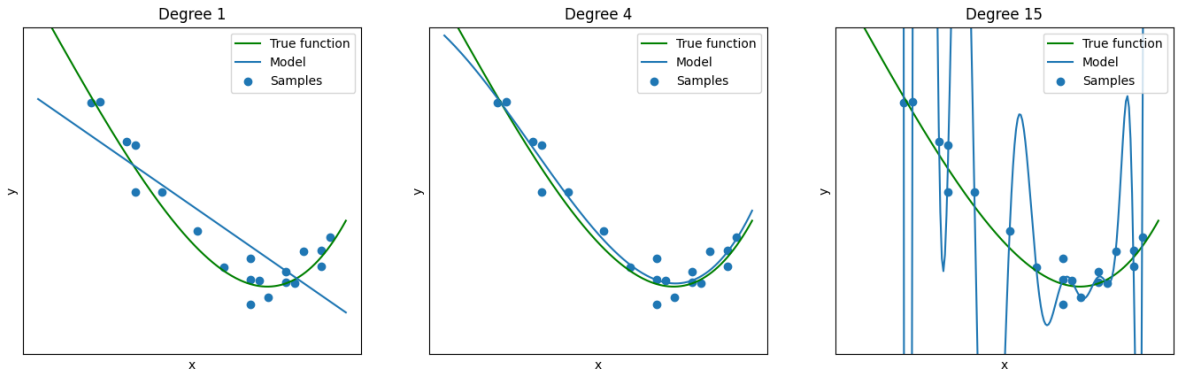


Figure 13 Here are three examples of different hypothesis spaces. The impact of this decision is known as the bias-variance tradeoff. If the space is too large (picture on the right) it may lead to satisfactory results on the training se but bad generalization (overfitting). If the space is too small (picture on the left), it gives bad results for the training dataset and for generalization.

4.1.1 Loss function

After defining $\mathcal{H}$ it is needed to define a loss function $\mathcal{L} : \mathcal{H} \to \mathbb{R}$ which assigns a loss to each $h \in \mathcal{H}$. This loss describes how well does h fit the given data D.
With a given loss function, the original problem now becomes an optimization problem:

$$\arg\min_{h\in\mathcal{H}} \mathcal{L}(h) = \arg\min_{h\in\mathcal{H}} \frac{1}{n}\sum_{i=1}^{n} l(x_i, y|h)$$

Where $\mathcal{L}$ is the loss for the hypothesis h given D, and l is the loss of a single data pair (x_i, y) given h.

Some classically used loss functions are the *Zero-One Loss*

$\mathscr{L}(h) \coloneqq \frac{1}{n}\, i = 1 n \, \delta h(x_i) \neq y_i$ (1) and the *Squared Loss* $\mathscr{L}(w) \coloneqq \frac{1}{n}\, i =$
The Zero-One Loss function is

$$\mathcal{L}(h) \coloneqq \frac{1}{n}\sum_{i=1}^{n} \delta_{h(x_i)\neq y_i} \qquad (1)$$

Where δ is the Dirac delta function

$$\delta_{h(x_i)\neq y_i} = \begin{cases} 1, & h(x_i) \neq y_i \\ 0, & otherwise \end{cases}$$

In general, the Zero-One Loss function is rarely used since is non-differentiable and non-continuous, which leads to difficulties with optimization algorithms.

The Squared Loss function is:

$$\mathcal{L}(w) \coloneqq \frac{1}{n}\sum_{i=1}^{n} (h(x_i) - y_i)^2 \qquad (2)$$

The Squared Loss function tends to punish more large errors and smooth out smaller ones, normally meaning that when searching for $h \in \mathcal{H}$ it will prefer functions with many small errors rather than one with a few large ones.

4.2 UNSUPERVISED LEARNING

Unsupervised learning, unlike supervised learning, does not know beforehand the labels corresponding to its initial dataset. In other words, Y is equivalent to the empty set. Meaning that this kind of algorithms must explore the inherent patterns, structures or relationships within the data to gain insight and make meaningful predictions.

Unsupervised learning utilizes mathematical techniques to model the structure of the data, searching for similarities or dissimilarities between data samples and grouping them accordingly. In general algorithms used in unsupervised learning tend to cluster datapoints based on their proximity in their feature space.

A common approach to unsupervised learning is clustering, which partitions data into clusters based on their similarity. Using an objective function the algorithm tries to minimize the values between data points within a same cluster, while maximizing the value if the compared data points belong to different clusters. For example, k-means clustering works by using cluster centroids, and iteratively assigning data points to selected clusters by minimizing the sum of squared distances between each point and its assigned cluster centroid.

An important technique used in unsupervised learning is the reduction of dimensionality of data. Since complex data can be hard to analyse, dimensionality reduction aims to capture the essential information of the data while reducing its complexity. A widely used approach is principal component analysis (PCA), which identifies orthogonal tensor in the data, called principal components, that capture the maximum variance within our dataset. Bringing our data to lower dimension using these principal components it becomes easier to analyse.

In many deep neural networks unsupervised learning can be seen as a layer with which the network can extract implicit features of the data and propagate them forward.

4.3 REINFORCEMENT LEARNING

Reinforcement learning is a specific type of machine learning which requires a dynamical system to interact with an agent, such that given a set of decisions by this agent, it can maximize a cumulative reward.

An agent or decision-making entity is the object which interacts with the environment at a given time and takes actions based on the state of the environment at that moment.

An environment is a “world” in which the agent operates.

A state (s) is a representation of the environment at a given time, capturing the information relevant for decision making on part of the agent. And the combination of all possible states is known as state space $S = \{s_1, \dots, s_n\}$.

An action (a) is the choice or decision made by the agent in response to a given state. The action space is the set of all possible actions on a given state $\mathcal{A}(s_i) = \{a_1, \dots, a_m\}$. Upon taking an action a state transition ($s_i \xrightarrow{a} s_j$) is executed. State transitions define the interactions between the actor and the environment. In general, to model non-deterministic cases state transitions are defined as probabilities $p(s_j|s_i, a_k)$ which describe that give our action a_k from state s_i what is the probability to end in the state s_j.

A reward is the feedback given to the agent based on its actions, it can be ether positive, negative or even neutral signals, guiding the agents learning process. It generally is described as the probability to receive a reward r given the execution of an action on a certain state $p(r_j|a_j, s_i)$.

Policy (π) is the strategy or behaviour utilized by an agent to select an action in different states. Specifically, is the probability of the likelihood of taking any given action on a given state $\pi(a_k|s_i)$.

The value function is the expected cumulative reward an agent expects to receive from taking a specific action in a particular state, following a specific policy.

If the probability for a state transition and the probability for rewards have the Markov property, meaning that they only depend on their previous state and not their total history. Then we have a Markov Decision Process (MDP).

The objective of reinforcement learning algorithms is to learn policies such that it maximizes the cumulative reward.

4.4 ARTIFICIAL NEURAL NETWORKS AND DEEP LEARNING

Artificial neural networks or simply Neural networks (NN) are a prominent type of machine learning model inspired by the structure and functionality of the human brain. NNs are a net of interconnected nodes, nominated artificial neurons or simply neurons (a), which process input data and generate an output. Each neuron has an "Activation" value between 0 and 1 which determines the effect it will have on its connected neurons on later layers. A layer is a set of "parallel" neurons, neurons on a same layer are written as $a_i^{(t)}$, where i is the index identifying the neuron on layer t. Every model consists of an input layer, an output layer and at least one hidden layer (an example can be seen in Figure 14). The structure of NN is based on acyclic weighted graphs ($G = (\mathcal{W}, \mathcal{N})$)

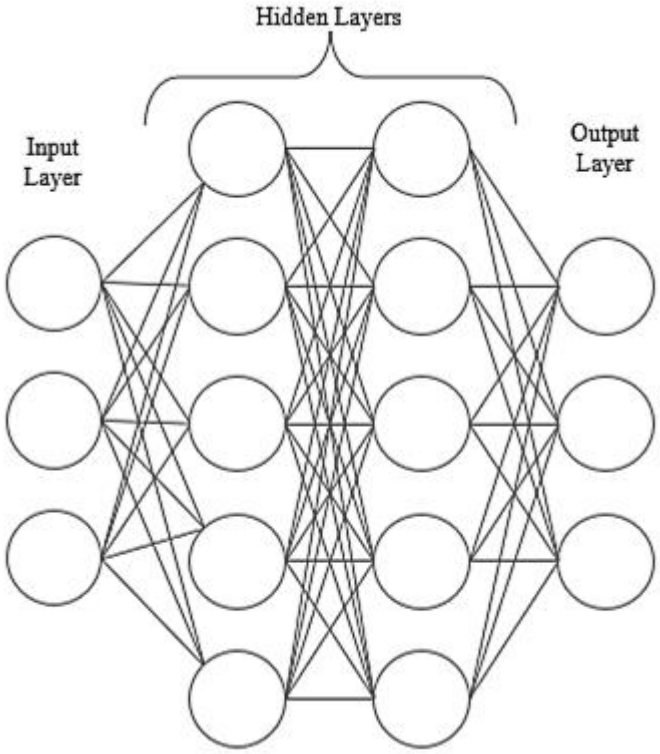


Figure 14 example of a MLP

and is loosely based on the activation of neural circuits. Specifically NN have a forward direction such that neurons can only be connected to successive layers. These connections have a weight w_{ij} connecting neuron $a_i^{(t)}$ to a neuron $a_j^{(t+k)}$, if for all neurons in a network $k = 1$, then its called a feedfoward network, else is called a recurrent network. For a neuron to activate is commont to have a threshhold value per each neuron called a bias (b_j). The total activation of the neuron is given by a activation function (such as the sigmoid function) evaluated on all the weights of the previous neurons and the current neuron bias, this is done for each neuron at each layer, this process is called forward propagation. It can be represented as the application of the activation function to the product of a weight matrix times the layer neurons represented as a vector plus a vector representig that layer respective biases.

$$a^{(i)} = \sigma^{(i)}(\mathcal{W}^{(i)}a^{(i-1)} + b^{(i)})$$

Where σ is our activation function, $\mathcal{W}$ is the matrix representing all the weights connecting the previous neurons to the current ones, $a^{(j)}$ is the vector of neurons on layer $i-1$ and b is our bias vector in layer i. For simplification of later scripture, $\sigma_i^{(k)}$ will be the output for node i from the layer k

Although the selection of activation functions σ can seem arbitrary, networks tend to prefer a small subset of well-behaved functions, the most utilized are:

- The Rectifier Linear Unit (ReLU) the function $f : \mathbb{R} \to [0, \infty)$ with form:

$$f = \max\{0, x\}$$

- The Leaky ReLU the function $f : \mathbb{R} \to \mathbb{R}$ with form:

$$f = \max\{0, x\} + \alpha \min\{0, x\}$$

- The Sigmoid the function $f : \mathbb{R} \to [0,1]$ with form:

$$f = \frac{1}{1 + e^{-x}}$$

- The Hyperbolic tangent the function $f : \mathbb{R} \to (-1,1)$ with form:

$$f = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

4.4.1 Deep learning

The description of "deep" learning refers to the use of multiple hidden layers in a network. Deep learning itself is the training of a deep neural network. As explained in 4.1.1 we can use a Loss function to evaluate the performance of a network. In the specific, given a dataset $D = \{(x_1, y_1), \dots (x_i, y_i)\}$, where x_i are our input data known as *instances* and belong to the set X, and y_i is our labels or reference output values known as *targets* belonging to the set Y. The final objective of a neural network is to model a composite parametric function $f_w : \mathbb{R}^n \to \mathbb{R}^m$. We can evaluate the performance of our network by means of our loss function $\mathcal{L}$. We specifically describe the loss function of a neural network as a function $\mathcal{L} : \mathbb{R}^n \to \mathbb{R}$ that quantifies the difference between two or more values. For example, a classic example is using the Mean squared distance between every output value γ_i and the corresponding target values y_i, after which from these results the total loss is calculated as the average of all of them.

Before describing the mechanisms used to tune NNs, is important to remark the Bias-Variance trade-off. This trade-off describes the relationship between a model's complexity, the accuracy of its predictions, and the performance of its predictions on previously unseen data, in Figure 13 we show this trade-off. In general, as we increase the size of our model (increase the number of tuneable parameters) it becomes more flexible, and it can fit better a training dataset. However, for

more flexible models, there tends to be great variance on the model estimation parameters, meaning that using different initial values initiating a model can lead to radically different results.

4.4.2 Loss optimization

As stated before, we can evaluate the performance of our network using a loss function, but more importantly we can adjust the weights in our network based on said loss function. Practice through which this is done is called *backpropagation*, a method in which it is tried to minimize the expected value of the loss function. This is done using the gradient of the loss function updating the influencing parameters in such a way that we find a local minimum.

We update our weights and biases values from a state t to a successive state is formulated as

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_{\theta} \mathcal{L}(\theta)$$

Where in this specific case θ refers to the combination of all the weights and biases of our network, and η is the learning rate of the network.

The mayor problem when training a multilayer feedforward network is improving the internal representation, i.e. the values the θ parameters should be. Since hidden layers don't have a target output, they are only updated based on how they affect future layers, but this can be determined since the values of any layer is dependent on a function based on the previous layer inputs.

The derivation of backpropagation is a combination of the chain rule and product rule.

For our study case we define the derivative of our activation function of layer k as $\sigma^{(k)}$ as $\sigma'^{(k)}$, and will the output from a neuron i as o_i. To simplify our operations and not repeat all calculations for the biases, a bias $b_i^{(k)}$ for the neuron i in layer k will be considered as a regular weight $w_{0i}^{(k)}$ with a constant output for $o_0^{(k-1)} = 1$ for layer $k-1$. Meaning that

$$a_i^{(k)} = b_i^{(k)} + \sum_{i=1}^{r^{(k)}} w_{ji}^{(k)} o_j^{(k-1)} = \sum_{i=0}^{r^{(k)}} w_{ji}^{(k)} o_j^{(k-1)}$$

Where $r^{(k)}$ is the total number of neurons on layer k.

The derivation of the backpropagation algorithm begins by applying the chain rule to our loss function

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(k)}} = \frac{\partial \mathcal{L}}{\partial a_j^{(k)}} \frac{\partial a_j^{(k)}}{\partial w_{ij}^{(k)}}$$

Where $a_j^{(k)}$ is the activation (product-sum plus bias) of the neuron j in layer k, before being passed to the activation function σ. This just implies that the change in the loss function due to the weights,

is proportional to the changes in our loss function due to our activation $a_j^{(k)}$ times the changes in the activation due to the weight $w_{ij}^{(k)}$.

The term $\frac{\partial \mathcal{L}}{\partial a_j^{(k)}}$ is the **error**, described as:

$$\delta_j^{(k)} = \frac{\partial \mathcal{L}}{\partial a_j^{(k)}}$$

And the term second term can be expanded as:

$$\frac{\partial a_j^{(k)}}{\partial w_{ij}^{(k)}} = \frac{\partial}{\partial w_{ij}^{(k)}} \left(\sum_{l=0}^{r^{(k)}} w_{lj}^{(k)} o_l^{(k-1)} \right) = o_i^{(k-1)}$$

Meaning that the partial derivative of the loss function $\mathcal{L}$ with respect to a weight $w_{ij}^{(k)}$ is

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(k)}} = \delta_j^{(k)} o_i^{(k-1)}$$

The calculation of $\delta_j^{(k)}$ can be expanded to use the expected values for the next layer instead of simply the final value $\hat{y}_j$. Meaning that the error term propagates backwards through the network, from the output layer up to the input layer.

Given this we can calculate the loss on our error as follows:

$$\delta_j^{(k)} = \frac{\partial \mathcal{L}}{\partial a_j^{(k)}} = \sum_{l=1}^{r^{(k-1)}} \frac{\partial \mathcal{L}}{\partial a_l^{(k+1)}} \frac{\partial a_l^{(k+1)}}{\partial a_j^{(k)}}$$

For layer $1 \leq k < m$, where m is the number of total layers of the network.

Further develop this equation by using the activation function $\sigma^{(k)}(x)$ for the hidden layers such that

$$\frac{\partial a_l^{(k)}}{\partial a_j^{(k-1)}} = w_{jl}^{(k)} \sigma'^{(k)}(a_j^{(k-1)})$$

And after changing the first term for the error term in or original error term we get a final equation:

$$\delta_j^{(k)} = \sum_{l=1}^{r^{(k+1)}} \delta_l^{(k+1)} w_{jl}^{(k+1)} \sigma'^{(k+1)}(a_j^{(k)}) = \sigma'^{(k)}\left(a_j^{(k)}\right) \sum_{l=1}^{r^{(k+1)}} w_{jl}^{(k+1)}\ \delta_l^{(k+1)}$$

Finally, the partial derivative of the loss function $\mathcal{L}$ with respect to the weight in the hidden layers $w_{ij}^{(k)}$ for $1 \leq k\ < m$ is

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(k)}} = \delta_j^{(k)} o_i^{(k-1)} = \sigma'^{(k)}\left(a_j^{(k)}\right) o_i^{(k-1)} \sum_{l=1}^{r^{(k+1)}} w_{jl}^{(k+1)} \delta_l^{(k+1)}$$

This equation is known as the backpropagation formula. And describes how the error $\delta_j^{(k)}$ is dependent on the errors of following layer $\delta_i^{(k+1)}$. Showing that the error moves backwards in the network.

To utilize this formula for the whole network we just need to specify that the final layer error term. This final term comes from $\mathcal{L}'(\hat{y}, y)$ where $\hat{y}$ is the prediction of our network and can be written as $\sigma_o(a_i^{(m)})$ and y is our target value. Here as an example, we show what this term may be like using the average square distance between $\hat{y}$ and y.

$$\mathcal{L} = \frac{1}{N} \sum_{d=0}^{N} \frac{(\hat{y}_d - y_d)^2}{2} = \frac{1}{N} \sum_{d=0}^{N} \mathcal{L}_d$$

Where $\mathcal{L}_d$ describes each output of the network. These gives us the partial derivative as

$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{(m)}} = \frac{1}{N} \sum_{d=0}^{N} \frac{\partial \mathcal{L}_d}{\partial w_{ji}^{(m)}}$$

And for each output we can calculate this as

$$\frac{\partial \mathcal{L}_d}{\partial w_{id}^{(m)}} = \delta_d^{(m)} o_i^{(m-1)}$$

Where in this case since we are using the squared distance as an example we have

$$\mathcal{L}_d = \frac{(\hat{y}_d - y_d)^2}{2} = \frac{1}{2}\ \sigma_o\left(a_d^{(m)} - y_d\right)^2$$

Then we can get the specific error for each output as

$$\delta_d^{(m)} = \frac{\partial \mathcal{L}_d}{\partial a_d^{(m)}} = \left(\sigma_o\left(a_d^{(m)}\right) - y_d\right) \sigma_o'(a_d^{(m)})$$

Meaning that

$$\frac{\partial \mathcal{L}_d}{\partial w_{id}^{(m)}} = \delta_d^{(m)} o_i^{(m-1)} = \left(\sigma_o\left(a_d^{(m)}\right) - y\right) \sigma_o'\left(a_d^{(m)}\right) o_i^{(m-1)}$$

4.4.3 Applying backpropagation

Backpropagation can be applied using 4 steps, assuming an adequate learning rate η and a random initialization of all parameters θ.

1. Calculate the forward pass for each training sample (x , y) store the outputs $\hat{y}$, $a_j^{(k)}$ and $o_j^{(k)}$ for each neuron j in every layer in our network, starting from the input layer and progressing in order until reaching the output layer.
2. Calculate the backward pass for each pair (x, y) and store the result from $\frac{\partial \mathcal{L}_d}{\partial w_{ij}^{(k)}}$ for every weight in the network.
    a. Evaluate the error terms δ_d in the final layer.
    b. Backpropagate the error terms to the hidden layers in the network, until every error term has been calculated for every neuron for every output *dimension* in $\hat{y}$.
    c. Evaluate the partial derivatives from all $\mathcal{L}_d$
3. Combine the separate gradients just calculated $\frac{\partial \mathcal{L}_d}{\partial w_{ij}^{(k)}}$ to get the total gradient $\frac{\partial \mathcal{L}}{\partial w_{ij}^{(k)}}$ with this get a pooling function to reduce it to a single value (generally averaging the values is a good option)
4. Using η as a scaling value, update the weights in the network following the opposite direction of $\frac{\partial \mathcal{L}}{\partial w_{ij}^{(k)}}$

4.5 INTRODUCTION TO GEOMETRIC NEURAL NETWORKS

For this thesis a requirement was set from the beginning: "the use of *AI tools* to improve remeshing in some way", this statement lead to the investigation of NNs more specifically we investigated *deep learning mesh reconstruction and generation methods* (for a detailed survey we suggest [40], [41] and [42]). We focused on learning the type of networks used for geometric processing and what their limitations and benefits are. First, we will give a few definitions of what geometric neural networks are and some important concepts used regularly on their design.

So far, we have focused on networks which work with Euclidean data. Which is data represented in multidimensional linear spaces that obeys Euclidean postulates ((1) A straight line segment can be drawn joining any two points, (2) any straight line segment can be extended indefinitely in a straight line, (3) given a straight segment, a circle can be drawn having this segment as its radius, (4) all right angles are congruent, (5) if two lines intersect a third in such a way that the sum of the inner angles on one side is less than two right angles, then the two lines must intersect each other if they are extended far enough.). By consequence means it can be measured by Euclidean

distances. Unfortunately, we wish to work on "curved" surfaces, which means that we need to work with non-Euclidean data.

Since we aim to work with meshes, we suffer from the called *dimensionality curse.* Meaning that when working in higher dimensions the number of parameters needed to have equivalent results grows exponentially. To reduce this issue, a normal approach used is to use strategies such as principal component analysis, or feature learning. To avoid discarding important information, normally data is transformed into a geometric structure called **Geometric prior.** Some examples of uses for geometric priors are:

- Processing independently of positional shifts, common for Convolutional neural networks (CNNs).
- Processing on spherical surfaces independently of rotation, used in spherical CNNs.
- Processing data independently of isomorphisms, common for graph neural networks.

For geometric deep learning the two main priors used to augment the data are:

- Symmetry: respected by functions which leave the object invariant, they must be invertible, composable and should contain the identity function.
- Scale Separation: the function should be stable under controlled deformation of the domain.

Although these principles are not hard definitions, it's been shown that following them provides noticeable better results on DNNs. This all can be seen as data augmentation, since many of these priors can be induced at the time of training.

4.5.1 Convolutional Neural Networks

A convolutional neural network or CNN is a special kind of deep network which is used with multidimensional data, specifically they are widely used when working with images and video.

In CNNs we can find some specialized layers, whose objective is to abstract features implicit in the data. These layers are convolutional layers, which use filters to abstract characteristic from the initial data, and pooling layers, which down sample their input data into a lower dimension. Normally convolutional layers are followed by a pooling layer, and after a couple iterations of them, their result is fed to a Multilayer Perceptron (MLP) [4] to realize the desired predictions on the working data.

We can think of a CNN as two separate networks, one dedicated to feature learning, while the second one is used for a final objective, for example classification.

Feature learning works using a convolution operation, normally using a linear operator such as the kernel function to extract features. The kernel function is also called a filter. Suppose we have an image I with dimensions $h \times w \times c$, where h is its height, w its width and c are a set number of channels. To this image a filter k is applied. A filter is a tensor of dimensions $n_1 \times n_2 \times n_c$ (where

[4] MLP: a fully connected feed foward neural network, normally not too big nor too deep.

the dimensions of n_c must match the dimension of c). The convolution works as if moving k from left to right on top of I, multiplying and summing the intersected section of I by its own values. This operation when applied to every pixel on the image creates a feature map F, with dimensions $(h - n_1 + 1) \times (w - n_2 + 1) \times 1$. These kind of filters focuses on the center of I, to mitigate this bias, normally zero padding is applied to the edges of I, so any value outside of the boundaries of I equals zero, making the final dimensions of F $h \times w \times 1$.

In practice, every "pixel" of F can be described by the formula

$$F_{[i,j]} = (I \otimes K)_{[i,j]}$$

And any ij^{th} entry is given as

$$f_{[i,j]} = \sum_{x}^{h} \sum_{y}^{w} \sum_{z}^{c} K_{[x,y,z]} I_{[i+x-1,j+y-1,z]}$$

In general, after a filter map has been created, to it is added a bias matrix with equal dimensions, and to its result an activation function is applied. Meaning that we can describe the output of our convolutional layer as

$$\sigma(I \otimes K + b)$$

To further simplify these features, a pooling layer is used. Where simply a pooling function $\sigma_p : \mathbb{R}^{m \times n} \to \mathbb{R}$ is applied as its activation function, $n \times m$ is the dimension of the kernel in charge of selecting the elements for pooling. Pooling functions are applied by a selected patches P from F or the respective $(n \times m)$ matrix. Some classical examples for pooling functions include maximum pooling, sum pooling and average pooling.

4.5.2 State-of-the-art review – meshing methods based on deep learning

When investigating the state of the art for intelligent mesh generation, the algorithms are normally separated based on three parts: Their techniques, their acceptable input data types and their output mesh units (be it triangular meshes, quad meshes, hexahedral meshes, among others), furthermore for quad meshes it is classified into direct and indirect methods based upon whether an intermediate representation is needed.

Some direct methods include:

[43], proposed a method using self-organizing finite element tessellation guided by a NN which deforms a simple initial mesh into a desired shape, by iteratively moving vertices reducing an error function using the silhouette of both meshes. [44] proposed a NN for automatic finite elements mesh generation, using an advancing front algorithm using node insertion predicting both position and connection types, the mesh boundary is advanced iteratively. [45] created MGNET, a NN model which takes boundary curves data as input and generates meshes with the desired number of cells, given a set of meshing rules. Unfortunately, direct methods, although convenient, can only manage simple planes or surfaces.

Indirect methods:

[46] Parametrize a model into simple shapes to easily learn special shape representation, they describe it as: *"a parametric template composed of Coons patches*". Given a raster image, the system infers the parametric surfaces and realizes an input in 3D, after which a quad mesh is generated following a given template, an easy example is the Boolean addition of the given coons. [47] created Sketch2PQ. Using strokes, depth sample and visible and occluded regions masks induced from a sketch input, Sketch2PQ extracts a direction field and B-spline surface from the initial sketch as an intermediate model representation. A quad-mesh is generated using geometry optimization of the B-spline surface. [48] generate a frame field from a trimesh, which is later used to guide parametrization for a mesh synthesis algorithm. This implicit frame field is derived from a local and global analysis of the geometry of a given type of meshes, meaning that given a dataset of topologically similar quad meshes it learns its shape and predicts the corresponding structure in a new trimesh.

4.5.3 Network architecture reviews

An analysis of several neural networks and their features was realized, checking based on data type, size, system requirements and expected results.

In general, the networks used for geometry processing are intended to modify existing vertices to obtain the desired deformation of a given model. [49] Show these principles using neural networks as tools to modify existing meshes. A simple approach to this is implemented in Pytorch3D [25], a library implementing a network capable of analyzing the error between shapes based on silhouette rather than coordinates. Where iteratively the network modified a single mesh trying to obtain a desired silhouette.

The trouble found following these kinds of methods is that they are useful when evaluating the resulting meshes but are rather inconvenient when trying to create new topologies from scratch. A possible application of this approach could be the fitting of a correct topological mesh to a similar shaped mesh.

Following the initial implementation in [48] we studied the original PointNet and SpiralNet.

PointNet is a point cloud-based analysis network usually used for classification and segmentation. This architecture is based mostly on 3 main modules: a pooling layer to aggregate information, a local and global information combination structure and two joining alignment networks to align input point and input features. Following this initial design PointNet++ and PointNeXt build up on it, improving performance by first linearizing the use of the modules and repeating their implementation and adopting data augmentation techniques on input data. By adding small amounts of noise to each point and including random changes in properties of the point clouds to remove implicit biases existing in it, plus they avoid loss of initial information by propagating it to later layer of their networks.

SpiralNet and SpiralNet++ are based on the creation of a 3d kernel based upon the existing surface to be analyzed. Starting from an initial vertex V, a set of k vertices is selected as its kernel, these

vertices start from a random vertex attached to V, from there in a clockwise direction with respect to V a new vertex is added (following as closely as possible the direction of the rotation). This process creates the bases for a 3D convolutional layer. The problem found with SpiralNet and SpiralNet++ was their inflexibility of input data, as by their design a regular number of vertices was needed for training of different figures. The use of them as singular modules proved too expensive to be used in practice, besides anisotropic meshes tended to have worse results around high-density areas as they covered a smaller surface.

We also reviewed MeshNet [50], a neural network designed with meshes as input data. The basic architecture of mesh net is separated into an initial network for feature extraction separated into two networks, one for spatial descriptors and one for shape descriptors, after which comes a set of mesh convolution layers, and finally a MLP is used to remap the result to usable outputs. The Spatial descriptor uses the barycenter of each triangle to describe its position; it's used as the input of a small network to describe the position of the vertices of faces. The structural descriptor is a topological feature analysis network, where the network is fed the topological characteristics of each face and realizes a convolution operation of it and its closest neighbors. Finally, the mesh convolution block is a network where the neighborhood of each triangle is expanded using the predicted features of neighboring triangles, it can be associated to a pooling layer in classical convolutional networks.

Lastly the other important NN studied was DiffusionNet. Describe as discretization agnostic, diffusion net allows for irregular data to be used as input for the training. Working using diffusion as convolutional layer, defined within a range applying the LaPlace operator on the existing surface. The biggest setback it has is that it is initially slower, due to the need to calculate the gradient along the surface among other regular parameters, which for training is rather significant. But it's been tested to provide better results with less memory use, and these initial parameters can be calculated beforehand such that their calculation time is reduced to a single cycle.

5 PROPOSED PROJECT

Our main objective is to convert a high-resolution triangle mesh T into a quad dominant mesh or a pure quad mesh which has a topological structure as close as possible as those used in animation while maintaining the original trimesh shape. Importantly the design of the network should work with input meshes with hundreds of thousands of vertices as the application of it should be guiding a remeshing of sculpted meshes which can have up to millions of vertices.

To this point we had to evaluate a dataset which had a suitable number of samples to train our networks and was approved as having a good topology according to experts in the field.

Once a suitable database was selected, we prepared the data for training, creating the ground truths (or functions to calculate them on the fly), we designed a way to create corresponding trimeshes to each quad mesh in the database, we selected a loss function, and we designed and implemented a NN.

The selected approach to evaluate the network was the prediction of a frame field constructed based on the topology of the ground truth quad-meshes. This approach was selected due to extensive research on similar topics such as cross fields and meshing using principal curvature directions. The creation of a frame field leads neatly to an evaluation using a network as it can be judged on the closeness of the prediction to a ground truth frame field, without the expensive step of the creation of a new mesh for each evaluation, further testing proved this to be the right approach as even without the creation of a new mesh, we faced hardware limitation imparted by the amount of memory available and time constraints due to the large execution times of several algorithms used. Other alternatives were considered but since frame fields lead to more flexible results they were considered as a prime candidate to judge the network.

The projection of the network is independent of the final prediction, meaning that even if we decided to use the network for other purposes, such classification or correspondence, we would've ended up using the same network architecture and just changed the output channel parameters.

5.1 SELECTED DATABASE - DFAUST

In this project we selected the DFAUST dataset [51] for our training basis. It was selected due to its size and range, with over 40.000 different models from 10 subjects in several different poses.

Although originally formed by triangular meshes, they were derived from the SMPL model [52], which was originally quad dominant mesh. A direct mapping from the new DFAUST created triangular meshes to an equivalent quad-dominant mesh was simple to create knowing this.

These models are mostly formed by quads, with some minor amounts of N-gons used to allow a better flow on the surface. The SMPL model has been studied and, according to the authors, over 70 experts on the field of animation have come to an agreement on the quality of the base model.

Working using methods of correspondence of 4D scans, a base model was deformed to fit selected subjects in different poses and motions. Meaning we have equal topologies for all the desired ground truth meshes independently of their geometry.

For training 80% of the meshes were randomly selected to train the network and the remaining ones were to be used for validation and testing. The total meshes were initially randomized and batched for future calculations, with 100 meshes for each pre-calculation batch.

Even though color data was available to use in the training as part of the training features, we explicitly decided not to use it, as for most meshes during the modelling pipeline color data would not be available.

Due to the large size of the data to be studied by our networks we need to adapt our solutions, as many networks are incapable of processing large data in a reasonable amount of time or due to a lack of memory (as seen in [53])

5.2 GROUND TRUTH FRAME FIELD

Our ground truth frame field is calculated per point on the surface using a ground truth mesh as a basis.

If there is a point P on our trimesh T the corresponding frame field value at that point will be calculated using P' which is the closest point on the corresponding pure quad mesh Q (the projection of P onto Q). P' is associated with a quad which will give us 4 directed half-edges e_1, e_2, e_3 and e_4 , where e_1, e_3 and e_2, e_4 are pairs of non-adjacent edges, and 4 distances d_1, d_2, d_3 and d_4, which correspond to the shortest distance d_i from P' to each edge e_i.

Our ground truth frame field will be described by the vectors u, v which are calculated as:

$$u = e_1 * \frac{d_3}{d_1 + d_3} + e_3 * \frac{d_1}{d_1 + d_3}$$

$$v = e_2 * \frac{d_4}{d_2 + d_4} + e_4 * \frac{d_2}{d_2 + d_4}$$

Since we are using the projection of P the possibility of a vertex match or edge match is considered low, and the corresponding frame field will be calculated as the weighted area sum of the adjacent frame field values of the corresponding quads around the vertex or edge.

5.3 PROPOSED ARCHITECTURE

The original idea was based on the concept of vertex painting or “skinning” used for animation. Where every vertex is given a weight corresponding to their transformation having to follow a *bone*. Having said this, the concept of skinning can be described as a segmentation problem of our

mesh. Using these predicted segments as inputs for a secondary network could have led to a good prediction if additional local information was given.

After comparing this strategy with the state-of-the-art, we decided to follow a model similar to that proposed by [48] and slightly inspired by [50]. We decided to capture the features of the global geometry of our mesh using a global network G and the local features for each triangle neighborhood using a local network L, after which they are both fed forward to a MLP which also receives a reference frame, the total network should predict our vector field at each barycenter of each triangle on our mesh. This allows us to leave the segmentation problem as an implicit part of the network (predicted by the global network) and allows us to have a smoother training regime.

5.3.1 Local network

The selected local network was PointNeXt [54], a cloud base neural network based originally on PointNet [35] and PointNet++ [55].

5.3.1.1 PointNet

The original PointNet was designed as a network capable of working with either point clouds or meshes as input data (using a combination of its vertices, barycenters and point sampling to create a point cloud related to the mesh). Each point in the point cloud is described by its coordinates plus a feature vector describing relevant information (like color, normals or curvature, among others). The original architecture of PointNet takes n points as input and is composed of three main modules: a pooling layer (as a symmetric function to aggregate information, by default max pooling), and two joint alignment networks aligning input points and point features.

The network was designed to comply with the required properties given by point clouds in mind 1.2. First to make the input invariant to permutations they use an approximation of a symmetric function which outputs a vector invariant to input order (as an example the sum is a symmetric binary function). They approximate the symmetric function by using an MLP on each point of the set and feed their results to a pooling function. The result is a set of abstract features f, described as the global signature of the initial input.

To aggregate local and global data a new layer is used, which takes as input the global features predicted by the last pooling layer and it's concatenated with a per point feature vector. This is used as input for an MLP which creates a new per point feature vectors aware of global features.

To deal with geometric transformations the joint alignment modules. These modules use a small network to predict an affine transformation matrix, this matrix is applied to the coordinates of input points or to the feature vectors depending on the module. Importantly these are two separate matrices and the one aligning the feature space tends to have much larger dimensions than the input points one. The feature alignment matrix is forces to approximate an orthogonal matrix to stabilize its results.

5.3.1.2 PointNet++

PointNet++ is an expansion of the initial proposal of PointNet. As PointNet doesn't consider data about the neighborhood of a point before doing its predictions. PointNet++ takes this into account and expands the concepts used in PointNet, using an approach inspired by CNN.

PointNet++ uses a hierarchical approach creating groups where a pooling function is used to describe them and later the groups are further grouped in larger neighborhoods with other groups. Each grouping is an abstraction level, and generally is used to concatenate applications of PointNet on each level. We can resume each *set abstraction layer* as three simple layers: a sampling layer, a grouping layer and a PointNet layer. The sampling layer defines the centroid of each group. The grouping layer selects the points which belong to each group. And the PointNet layer uses a small implementation of PointNet to encode local data as feature vectors.

The grouping layer is inspired by CNN as they use the Manhattan distance to create a kernel and group pixels. While for PointNet++ a space metric needs to be used.

This hierarchical approach allows for further flexibility using adaptive PointNet layers, which can adapt to regions with different densities. This can be done by either multi-scale grouping or multi-resolution grouping. Multi-scale grouping consists of applying grouping layers with different scales followed by corresponding PointNet networks to extract scale specific feature vectors. Multiple scales are concatenated together creating a multi-scale feature vector. Multi-scale grouping is expensive since it tends to use point net on large neighborhoods and multiple times per point set. The alternative approach is to use multi-resolution grouping, where the resolution of the feature vector is determined by the number of inputs in a given region. Adaptably changing the resolution scale based on the density of points, where the resolution of a region is determined as the level of abstraction. Meaning that for highly packed regions, a high level of abstraction can lead to better results, while for sparse regions even working directly with the points can be preferred.

For segmentation problems feature propagation must be done. The solution adopted by PointNet++ is hierarchical propagation with a distance-based interpolation. Which means, per each set abstraction layer an equivalent feature propagation layer is done, this feature propagation interpolates based on the original centroids and their distance to each point existent in each abstraction level.

5.3.1.3 PointNeXt

PointNeXt improves on PointNet++ mainly by improving the training strategies used and a small change to allow for better feature propagation, mainly the use of receptive field scaling and model scaling.

PointNeXt decides to improve on receptive field scaling by normalizing the values of the inputs on each abstraction level, allowing for smoother predictions. Model scaling on the other hand was already theoretically tackled by PointNet++. In PointNeXt they tested the influence on the addition of abstraction levels and the increase of input channels and discovered that it did not bring noticeable improvements, but did cause a noticeable drop of throughput. They showed that the

ideal size for their network is using four set abstraction blocks. To improve results, they create an Inverted Residual MLP (*InvResMLP*) block which is appended after each set abstraction block. Each InvResMLP block has a residual connection between the input and output to reduce the effect of a vanishing gradient and use multiple MLPs to reduce computation cost and still have improving pointwise feature extraction. The InvResMLP block are exemplified in the paper as a grouping layer, followed by an MLP, followed by a pooling layer and finishing with a chain of MLPs where the last one takes as inputs the initial point features as stated earlier.

The implementation of PointNeXt unifies the classification and segmentation models presented in PointNet++. Allowing for direct correspondence of set abstraction block (encoders) to the feature propagation blocks (decoders). They create each decoder such that the input channel size is coherent with that of the corresponding encoder. They also add an initial MLP at the beginning of the network to map input point clouds to higher dimensions. They presented four standard implementations of their network PointNeXt-S, PointNeXt-B, PointNeXt-L and PointNeXt-XL. Where each implementation has a difference on either initial channel size of the input MLP (C), or amount of InvResMLP block appended to each set abstraction block (B).

Training-wise the improvements implemented by PointNeXt include the use of a better optimizer (AdamW), improved learning rate scheduler and learning rate decay, and more advanced loss function (CrossEntropy with label smoothing). Plus, some training techniques such as the addition of random noise on the data at training time, data augmentation and random drop offs allowed to PointNeXt get better results.

For our purposes PointNeXt had a simple implementation for local geometry, using a similar schema as their own, we selected neighborhoods around each barycenter of the working trimesh and realized and created a used KNN algorithm to find the K nearest vertices to each barycenter to feed to the network. It was considered to explore the network as a graph and select the K nearest connected vertices, which although it could give slightly better results, the time to calculate these graphs would be too long for large values of K.

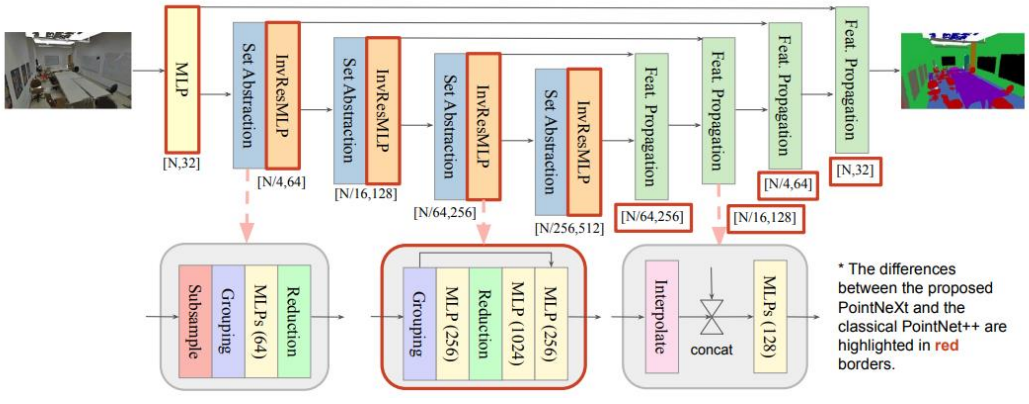


Figure 15 PointNeXt architecture as depicted by [54]

5.3.2 Global network

For our global network we decided to use DiffusionNet [53]. DiffusionNet is a network which uses diffusion as a main network operator. In their paper it is shown that a learned diffusion operation is sufficient to learn spatial data on surfaces.

DiffusionNet has three main parts: point wise MLPs applied pointwise on the input data emulating a scalar function of the feature channels, learned diffusion operation propagating information on the domain (the mesh surface in our case) and local spatial gradient features.

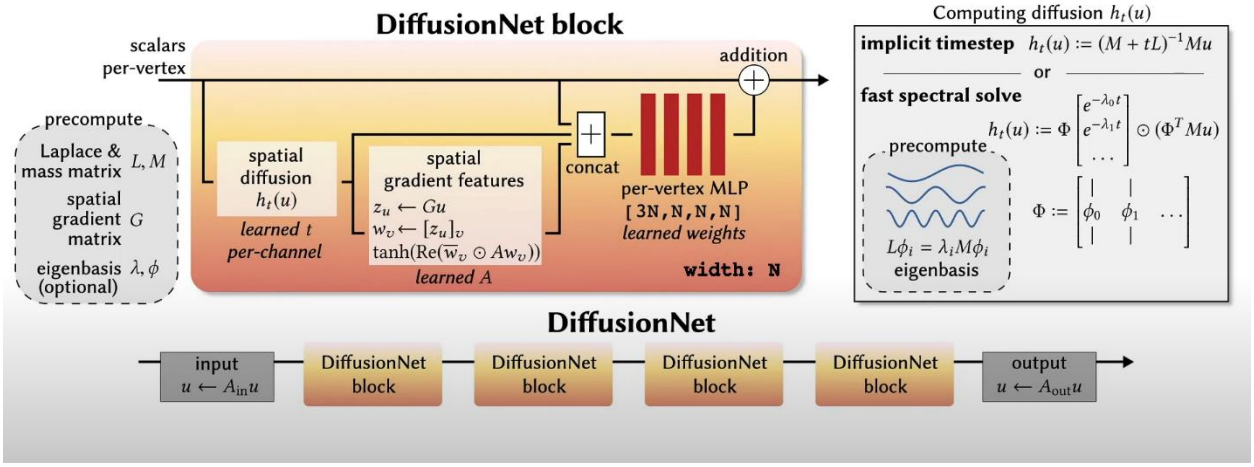


Figure 16 DiffusionNet architecture -DiffusionNet presentation video

The approach taken by DiffusionNet is to emulate a pointwise function using a pointwise MLP, with shared weights for the entire domain, but this approach doesn't allow for global data or multi point structures.

To capture global data DiffusionNet uses a learned diffusion. Diffusion on the continuous setting is modelled by the heat equation

$$\frac{d}{dt}u_t = \Delta u_t$$

Where Δ is the Laplace-Beltrami operator. The diffusion is represented by the heat operator, which is applied to some initial value u_0 a produces a distribution over time. This can be defined as

$$H_t(u_0) = \exp(t\Delta)\, u_0$$

Where 'exp' is the exponential operator. Diffusion can be seen as a smoothening process on the global geometry. When $t = 0$ diffusion should return the identity map, while when $t \to \infty$ it will be close to an average over the domain. DiffusionNet propagates learned features over the surface using the heat equation. This allows the final propagation to be largely invariant to the type of domain it works on. The Laplace-Beltrami operator can be replaced using the *weak Laplace matrix* L and the *mass matrix* M.

A learned diffusion layer h_t describes the diffusion of a feature channel u for a learned time t. In the network each channel applies $h_t(u)$ independently with a separately learned t per each channel. Diffusion can be seen as a pooling operation averaging value on the surface.

DiffusionNet demonstrated that diffusion can be a replacement for complex operations usually used to create kernels on surfaces, such as radial geodesic convolution.

Diffusion can be calculated using either a *direct implicit timestep* or *spectral acceleration.* The direct implicit time step is a direct approach which simulates diffusion using single implicit Euler timesteps.

$$h_t(u) \coloneqq (M + tL)^{-1}Mu$$

This reduces each diffusion operation to solving a sparse linear system. Using an implicit backward timestep is crucial as makes the schema stable, allows for global support and provides a good approximation of diffusion in a single step. This approach can lead back to solving dense linear systems, meaning it does not scale well to large problems (high resolution meshes).

The preferred approach to calculate diffusion in DiffusionNet is using a spectral acceleration approach. Leveraging closed-form expression for diffusion in the basis of low-frequency Laplacian eigenfunctions. Once an eigen basis is precomputed, diffusion can be computed for any time t via exponentiation applied to each element. For the matrices L and M the eigenvectors $\phi_i \in \mathbb{R}^V$ are the solutions to:

$$L\phi_i = \lambda_i M\phi_i$$

Corresponding to the first k smallest magnitude eigenvalues $\lambda_1, \dots, \lambda_k$.

Importantly the authors emphasize that DiffusionNet is not a spectral learning method, as the spectral coefficients are never used to represent filters or latent data. Spectral acceleration is only used to compute diffusion in a more efficient manner (as can be seen in Table 1).

The last important part of the projection of DiffusionNet are the *spatial gradient features.* These features are created of the inner product between pairs of feature gradients at each vertex (after applying learned scaling or rotation). The gradients are represented as 2D feature vectors tangent to each vertex. They are evaluated using a standard procedure. Using the normal of each vertex the closest neighboring vertices are projected onto the tangent plane of the vertex being evaluated. The gradient operator is computed in the tangent plane via least-square approximation of the function values at the neighboring points. These gradient operators form a sparse matrix $G \in \mathbb{C}^{V \times V}$. G is applied to a vector u to get the real values at each vertex. G is independent of the features and can be precomputed per shape. Scalar features can be deduced by evaluating an inner product between pairs of feature gradients at each vertex. For each channel u a spatial gradie $z_u \in \mathbb{C}^V$ can be constructed as a vector using the 2D gradients at each vertex.

$$z_u \coloneqq Gu$$

Method		***Small*** *752 verts*	***Medium*** *10k verts*	***Large*** *184k verts*
DiffusionNet *(spectral)*	*Pre:*	288ms	3.55 s	69.5 s
	Train:	19ms	25 ms	379 ms
	Infer:	7ms	10 ms	154 ms
DiffusionNet *(direct)*	*Pre:*	104ms	-	-
	Train:	329ms	-	-
	Infer:	81ms	-	-
MeshCNN *[56]*	*Pre:*	85 ms	1.13 s	-
	Train:	269 ms	2.97 s	-
	Infer:	194 ms	2.71 s	-
HSN *[57]*	*Pre:*	905 ms	162 s	-
	Train:	188 ms	1.08 s	-
	Infer:	68 ms	389 s	-
HodgeNet *[58]*	*Pre:*	*n/a*	*n/a*	-
	Train:	752 ms	7.61 s	-
	Infer:	645 ms	6.87 s	-

Table 1 Runtimes of DiffusionNet and other mesh-based methods compared with varying input mesh resolutions. Reported times provided by [53], describe a one-time preprocessing step (pre), a training evaluation with derivatives (train), and an inference evaluation (infer), each on a single specific resolution mesh. Entries mark by "-" were infeasibly expensive in time or memory usage. DiffusionNet of the reported networks is the only one capable of managing meshes of high resolutions.

Stacking all local gradient channels at each vertex created the complex feature vector $w_v \in \mathbb{C}^D$ which can be transformed into the corresponding real valued feature vector $g_v \in \mathbb{R}^D$.

$$g_v \coloneqq \tanh\left(Re\left(\overline{w}_v \odot A_{ij} w_v(j)\right)\right)$$

Where A is the learned square $D \times D$ matrix. This means that the i^{th} output at each vertex is given by the dot product $g_v(i) = \tanh\left(Re\left\{\sum_{j=1}^{D} \overline{w}_v(i) A_{ij} w_v(j)\right\}\right)$. The tanh operator is stated to be used to stabilize training but is not necessary. The use of A as a complex matrix directly allows for a richer representation on surfaces as implies rotation of the surface, if non oriented inputs are used (point clouds) the use of a real matrix for A shows to provide better results.

The architecture of DiffusionNet simulates a pointwise function at each vertex by applying an MLP, it uses learned diffusion for spatial communication, and spatial gradient features to model directional filters. This creates DiffusionNet blocks, which are used to create DiffusionNet. The network operates on a fixed channel of width D of scalar values, with each DiffusionNet block diffusing the features constructing spatial gradient features and feeding the results to an MLP. Residual connections are added to stabilize training, as well as a linear layer to get a desired output dimension. Outputs on faces or edges are interpolated from the corresponding limiting vertices.

The input features to calculate the scalar per vertex functions supported are either (*x, y, z*) coordinates or heat kernel signatures (HKS) (HKS describe intrinsic quantities related to the surface). (*x, y, z*) coordinates require data augmentation to improve results, such as random rotations and normalization, while HKS are computed from the spectrum of the Laplace operator and are invariant to rotations. Of course, if other data exists, such as color data, it can be used instead.

For our purposes we decided to use DiffusionNet with HKS as they are intrinsic values and would save us the time of realizing rotations to augment the data. Constants used for DiffusionNet are precalculated and cached before training.

5.3.3 Combining the networks

Our network architecture predicts a frame field value (two vectors u, v) on a per face basis using a combination of DiffusionNet and PointNeXt. We use a single block of both networks applied to the corresponding interest area (local neighborhood or total global geometry). And combine the using an MLP which takes as input a reference frame per each face, the global predictions and the local predictions. The reference frame is calculated as follows: per every face of the input trimesh we use its barycenter as an origin, the first basis vector will be directed using the first defined half-edge on the face, call it y_t. The second base vector is the normal from the face n_t and our final base vector is calculated as the cross product of Y_t and n_t

$$X_t = Y_t \times n_t$$

Finally, we normalize them. Giving us an orthonormal basis per each triangle of T.

Ideally a larger number of blocks would be used to predict larger local features but due to memory limitations we used a single block adjusted based on resolution. Our training data was remeshed making it regular over the whole surface, meaning that using a KNN algorithm works well forming local neighborhoods. The local neighborhoods were formed using the $K - 1$ nearest vertices to each barycenter. And the global prediction was done using the HKS input on the DiffusionNet block.

DiffusionNet was selected due to its tendency to work well with large resolution inputs, as our objective was to create a network capable of applying remeshing to very high resolution meshes. And PointNeXt was selected due to its flexibility, as the definition of local neighborhoods can be expanded to use connectivity (at the price of non-minor increase of computing time if not cached beforehand).

A major advantage of this design is its ability to handle large meshes and the ability to give extra features as inputs as we deem fit (either as part of the feature vectors on PointNeXt or as part of the inputs for the final MLP). Property that we took advantage of, by adding the principal directions on all faces as part of the features describing a face.

5.4 LOSS FUNCTION

Our loss function must determine the accuracy of our predictions with respect to our ground truths, since we are working with frame fields, we decided to use a loss function which would evaluate both vectors describing our prediction with respect to out ground truth. We evaluated our loss with respect to our vector modules and angles separately. And since we cannot assure that our predictions are in order, we realized a "combing" operation with respect to our ground truth, checking both options and using the best result. Our total loss was given by a module loss function $\mathcal{L}_m$ and an angle loss function $\mathcal{L}_a$. We work with our predictions $x = < u, v >$ we calculate their loss respect to our ground truth vector $y = \langle \hat{u}, \hat{v} \rangle$.

Besides that, we decided to add a scaling factor to adjust both the relevancy of which function and the scale at which they operate.

$$\mathcal{L}(x, y) = MS * \mathcal{L}_m(x, y) + AS * \mathcal{L}_a(x, y)$$

Where MS and AS are our module scaling factor and angle scaling factor respectively, and our loss of x with respect to y is given by the sum of our module loss plus our angle loss, both with x respect to y.

We calculated our module loss as the function:

$$\mathcal{L}_{m_{1d}}(w, \hat{w}) = \frac{\big||w| - |\hat{w}|\big|}{|\hat{w}|}$$

And our total module loss is given by the sum of both vector losses.

$$\mathcal{L}_m(x, y) = \mathcal{L}_{m_{1d}}(u, \hat{u}) + \mathcal{L}_{m_{1d}}(v, \hat{v})$$

And our angle loss or direction loss is calculated on the *Von Mises* distribution which approximated a normal distribution around a unit circle, it was selected due to its previous use on similar research papers. This allows us to formulate a loss for angles θ targeting $\hat{\theta}$ that is invariant with respect to rotations of 2π radians.

$$\mathcal{L}_{vm}\left(\theta, \hat{\theta}\right) = 1 - e^{k\,(\cos(\theta - \hat{\theta}) - 1)}$$

Since our frame can be seen as two independent directions with π symmetry each, we modify this function.

$$\mathcal{L}_{vm}^{2x}\left(\theta, \hat{\theta}\right) = 1 - e^{k\,(\cos(2(\theta - \hat{\theta})) - 1)}$$

We can further adapt this formula to work directly on our vectors using the double angle formula.

$$\mathcal{L}_{vm}^{2x}\left(w', \widehat{w'}\right) = 1 - e^{k\,(2(w' \cdot \widehat{w'})^2 - 2)}$$

Where both w' and $\widehat{w'}$ are both unit vectors.

This leads us to have a function which evaluated on both u and v gives us a total loss per triangle.

$$\mathcal{L}_a(x,y) = \mathcal{L}_{vm}^{2x}(u,\hat{u}) + \mathcal{L}_{vm}^{2x}(v,\hat{v})$$

Since we cannot ensure that our predictions are combed for our frame field, we evaluate both x and $\hat{x}$. Where $\hat{x} = < v, u >$. There giving us 2 partial losses $\mathcal{L}_1$ and $\mathcal{L}_2$.

$$\mathcal{L}_1(x,y) = MS * \mathcal{L}_m(x,y) + AS * \mathcal{L}_a(x,y) \qquad (3)$$

$$\mathcal{L}_2(\hat{x},y) = MS * \mathcal{L}_m(\hat{x},y) + AS * \mathcal{L}_a(\hat{x},y) \qquad (4)$$

Finally, our total loss is given by getting the minimum of our partial loss functions $\mathcal{L}_1(x,y) = MS * \mathcal{L}_m(x,y) + AS * \mathcal{L}_a(x,y)$ (3) and $\mathcal{L}_2(\hat{x},y) = MS * \mathcal{L}_m(\hat{x},y) + AS * \mathcal{L}_a(\hat{x},y)$ (4)

$$\mathcal{L}(x,y) = MIN(\mathcal{L}_1(x,y), \mathcal{L}_2(\hat{x},y))$$

After which to evaluate it over each batch we calculate the average of all $\mathcal{L}(x,y)$ calculated on each triangle.

6 TRAINING

In this section we will describe specifically what steps were necessary to follow for the implementation of the training of our network.

6.1 DATABASE PREPARATIONS AND SEGMENTATION

6.1.1 Ground truth

For our ground truth preparations, we had to remap the faces existing in the DFAUST models to match those in the original SMPL model. Fortunately, after some trials we managed to confirm that the trimeshes from DFAUST were simply the automatic triangularization that is done to quad-meshes by most 3D modelling software. Meaning that if the original quad mesh Q had F number of faces each face belonging to the corresponding trimesh T would be mapped using the module function. Where the index I of F' (a face of T) would point to the corresponding face F would be:

$$I = F'Mod(F)$$

This would map two faces from T to a single corresponding face in Q, after that we can simply check the order of the vertices to match create the corresponding face.

Since we are working with .OBJ files, and we discovered that the mapping is direct between the faces in Q and those in T, we can directly use the maps from Q to a new file Q_t which has the geometry of T and topology Q.

Now, at this point we have our quad-dominant meshes; we wish to transform them into pure quad meshes. We apply a single iteration of a simplified Catmull-Clark subdivision (on page 117), since one of the properties of the CC subdivision is that the created mesh will be a pure quad mesh Q'. We use the simplified version to maintain as much as possible the original geometry of the original meshes.

Given a corresponding trimesh used for training, for each barycenter on its surface, we can project it to its closest point on the newly created Q' and we can calculate the corresponding frame field vector pair following the procedure in 5.2.

6.1.2 Training data

To ensure that our training is not just learning the positions of the edges on the original trimeshes all training meshes were subjected to a remeshing algorithm.

First, we did 3 steps of simplified Catmull-Clark subdivision. After which we used a Voronoi remeshing algorithm, in which we first had to do clustering on our subdivided mesh, creating

vertex clusters weighted by area on our surface, after which we simply created faces following these clusters assuming them as the dual of a Delaunay triangulation, for a more detailed view check [22].

We checked if the resolution was enough, in the contrary case we realized a full CC subdivision until it was over a threshold and after we decimated the mesh if the number of faces was too high. Finally, to ensure that we had trimeshes for our inputs we saved them using the standard procedure for triangulation.

This procedure was executed to all meshes in our database, using a multithreaded approach to spare time. Our 10 subjects were selected as the input data for each thread, each thread read all the corresponding subject "movements", and for each movement read the amount of existing meshes that were used to represent it, with the equivalent of 1 mesh per file. Finally, we ended up with a corresponding trimesh per each ground truth quad mesh.

With this procedure we could not guarantee the exact same number of faces for all our meshes, so we had to adjust our network to make sure that a variation in the number of input faces was acceptable.
Since we are working with DiffusionNet, it was highly recommended to pre-calculate all relevant values to use it as a part of our network. Meaning for each training mesh, we calculated a mass matrix, 2 directional gradient matrices, and the corresponding HKS components. We saved all these pre-calculations in a cache for direct reading on the training. As a bonus since we knew we had to pre-calculate this data, we could use the time to pre-calculate the ground truth frame fields and the corresponding basis vectors, and for our latest models we also calculated the principal directions for all faces.

The total time varied depending on the desired resolution of the final mesh. Setting up the training data was around 1 week for the remeshing algorithm and a second week calculating all relevant data for meshes of ~ 60k faces for a high-resolution training, and approximately 10 days in total for meshes with 20k faces for lower resolution sped up training.

6.2 TRAINING PROCEDURE

Our training procedure was separated testing all the described networks and checking their results individually, after which we checked if our projected architecture had better results.

Along with the training one common obstacle found was the amount of available memory, both in RAM and in our GPU. Since we are working with rather large networks and with rather large input data, some options were explored to better suit our hardware limitations.

For starters, two approaches were considered: to batch one triangle at a time or to try and use larger batches, ideally a whole mesh at a time. Depending on the network the size of the batches was adjusted to try and use the most memory available without risking crashes. This is necessary since even with our best-case scenario (using full meshes as a batch), the enormous volume of data that we have available makes every epoch take a long time. Evaluating a "small" network we evaluated

approximately 4000 meshes in an 85-hour period, given that initially we assigned approximately 35 thousand meshes for training, each epoch would take around 750 hours or about 31 days. As these times were too long for complete training of multiple networks to compare results, we evaluated the best result from different implementations of the proposed base network on a period of up to 3 weeks. Which allowed us to fairly estimate the results of our different implementations based on period.

We had to take special care to check if our loss function was providing us with the best results. We modified its weights AS and MS and checked if the results improved. Particularly we set MS to 0, giving us notable better results for the predicted angles (which are our priority as they are the main factor guiding the directional field). We had a separate run specifically using MSE loss function to compare the efficiency of the custom loss function selected. The results revealed that use of a normal distance function without careful consideration for the order of the vectors let to result comparable with a random distribution leading to the conclusion that our custom function was the best approach.

6.2.1 Training algorithm

Our training setup was done with the following steps:

- The creation/ loading of the required networks.
- Loading an optimizer for the training (we used the AdamW optimizer as is the most used optimizer and most implementations of geometric neural networks seem to perform better with it).
- We initialize the network and the optimizer parameters. We started loading the network data if there was preexisting training of the same architecture.
- We selected a device (CPU or CUDA) and assigned it to all parameters and continually assigned it to all tensors created during training.

The training cycle went as follows:

- We set the networks to training mode.
- We load the data based on the cached inputs saved before training.
- We augment the data: we normalize the input meshes, we add a small amount of noise to each vertex randomly, and if necessary, we do a random rotation of the mesh (for DiffusionNet using HKS is not necessary).
- We create a batch of faces for the prediction.
- We realize a prediction.
- Calculate the loss.
- Realize backpropagation.
- Repeat for all meshes for a determined number of epochs.

Three base implementations were tried: 1- updating multiple times the local network per global prediction, 2- using a single update of the full network per batch where a batch is a reduced subset of faces of the full mesh, and 3- doing a single prediction for the full mesh.

The first implementation (Implementation A) consisted of constantly backpropagating the local network at each step and only using the backpropagation on the global network once per batch. This allowed us to have larger batches and track the error evolution at each step. Unfortunately, this approach proved to have hard to read progress and was hard to evaluate which modifications helped improve training. And since the global network was updated much less regularly, significant progress seemed sparse.

The second implementation (implementation B) set a specific number of triangles as a regular batch size and the global network would make a single prediction per batch. After which backpropagation can be done to the entire network after each batch.

The last implementation (implementation C) was our ideal case, where we could evaluate full meshes at the time with feature vectors for each face. But due to some of the tried networks using enormous amounts of memory for large predictions, it was not possible to implement on every model, having to default to one of the previous cases for those networks. Still for the models that were used that managed to use this last setup, the results were noticeably better.

For each mesh the global data was loaded once and was passed to the relevant parts of the network (the model and the loss function).

After each batch the loss function evaluated our results and uploaded its progress.

Finally, we update the model learning rate and the optimizer learning rate separately. After a selected number of batches or number of total processed meshes independent for each one.

6.2.2 Ablation study

The ablation study consists on validating our choices by training the network in a controlled environment such that if we “turn on” and “off” certain parts of the network the expected result would be worse than with the full network.

We first approached the problem confirming if the best approach was to use a CPU or GPU (CUDA 117) based approach. We tried two implementations (A and B) using simply two small MLPs for both the local and global networks, as it would provide a fair comparison and not require excessive amounts of memory just to confirm our expected prediction. The results were comparable for each epoch in values, but the training times were noticeably better for the GPU, with an increase of x25 speed. Following these results all following trainings were done using GPU.

Next, we evaluated different parts of the network both on their own and with controlled combinations to discover the best model, this studies were done using a cutting point for all models as otherwise slower models would not finish in time, it was decided that the cutting point for all models would be either 2 weeks of constant training or 200 epochs running using all 4000 curated training subjects.

We evaluated using PointNeXt using both a *s* and a *xl* sized models on their own using all 3 implementations (A, B and C). Having minimal difference between both models, where the lowest loss registered for the *s* model being 1.08167 and the lowest loss for the *xl* model being 0.93865. As the difference between both models was negligible to improve performance it was decided to use the *s* model as it uses less memory, and it proved to be slightly faster.

We evaluated DiffusionNet on its own using all 3 implementations (A, B and C) having an overwhelmingly better performance over PointNeXt, arriving to have a loss as low as 0.5771. All 3 implementations performed at a similar level, but implementation C was faster.

We finished the study testing if using DiffusionNet as the Global network and PointNeXt as the local network could provide better results, we used implementations A and B. In this case the network performed worse than DiffusionNet on its own, as it finished with an error of 0.61488 for implementation B and 0.973221 for implementation A. On Figure 17 we can see a distribution of the angle error calculated on each face of a testing mesh.

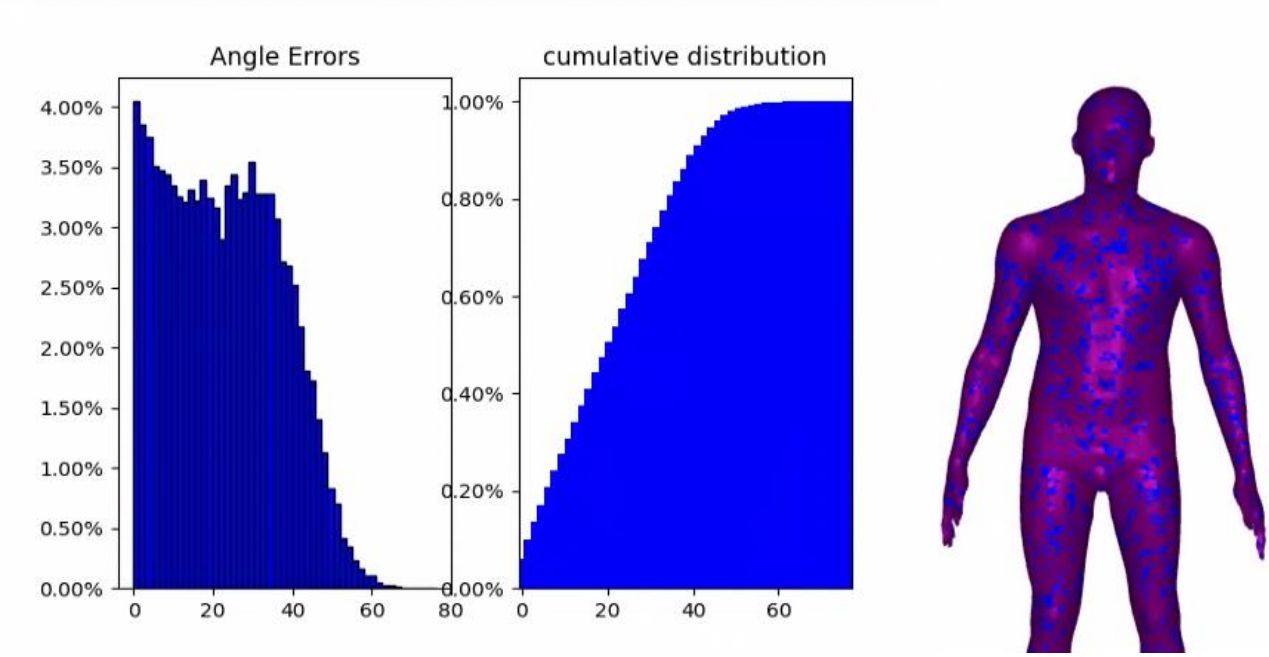


Figure 17 Representation of distribution of error for DiffusionNet + PointNeXt as a network using implementation B, blue faces have an average error <15`.

In Figure 18 we can see the best results provided by the implementation using DiffusionNet. These results are not accurate enough to drive a parametric remesh, so it was decided to advance with the experiments using only DiffusionNet and search to improve its performance.

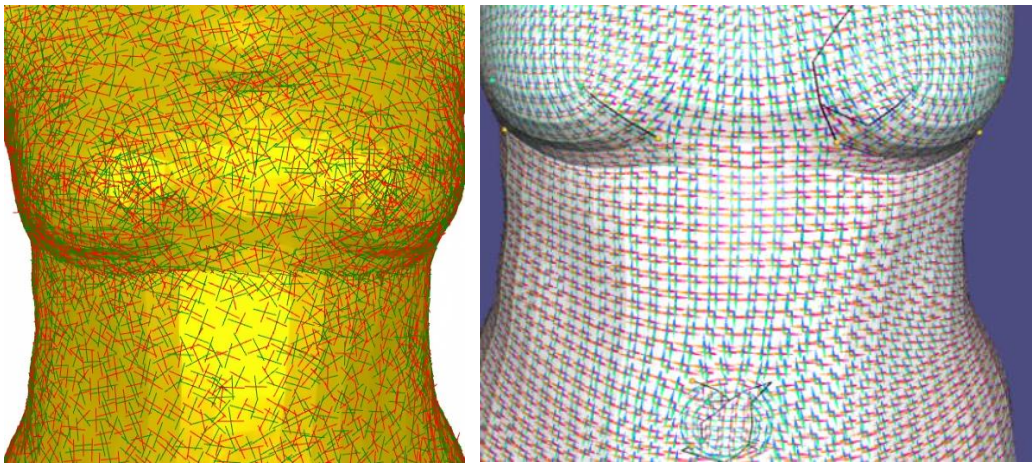


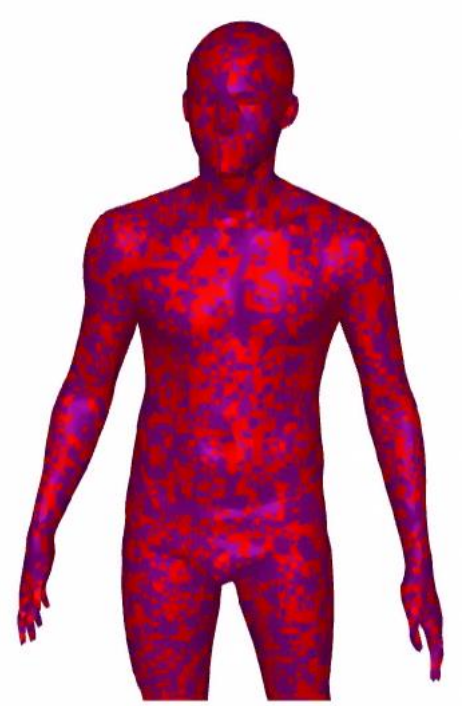
Figure 18 Comparison: DiffusionNet predicted unit frame field (left, python representation) vs ground truth frame field (right, cpp representation used to test remeshing algorithms)

Note: this figure shows the result of the best predictions we got up to that point. These results are still not usable for a meshing algorithm.

Given these results moving forward it was decided to use this model (DiffusionNet) as the benchmark on future testing.

6.2.2.1 *Validating the custom loss function*

We confirmed the efficiency of our custom loss function comparing its results to a training run using MSD on the predicted tensor with respect to the ground truth. Although the loss value got to a similar value (~0.56 for both functions), the error distribution for the MSD run was similar to that of a random distribution (Figure 19 and Figure 20).



Although the MSE as the loss function expedited the training tremendously, doing 200 epoch in 3 days, the results were not usable.

Figure 19 Figure representing the errors using the model trained with MSE as a loss function. Red faces represent those where the error is >45`

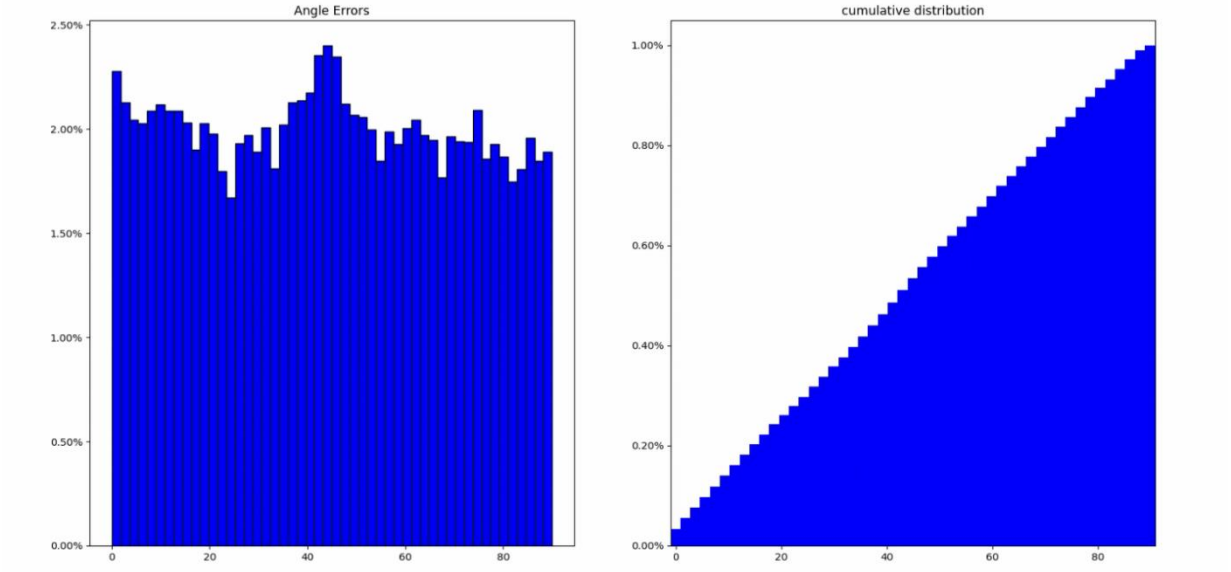


Figure 20 Distribution of the error on each face on a mesh using MSE as the training loss function

6.2.2.2 Using principal directions

For our best and most promising results we added the principal directions of each face (calculated using [59]) as part of the feature tensor. This although reached only similar levels loss (reaching 0.59001 as its best value at the end of the training). In Figure 211 we can observe the progress on the first epoch of the training using the principal directions as features. Observing the progress of the loss from this training (Figure 22) if our progress continues at a similar rate (excluding the first couple predictions) we would end up with ~70% of faces with an error <15` after 100 epochs.

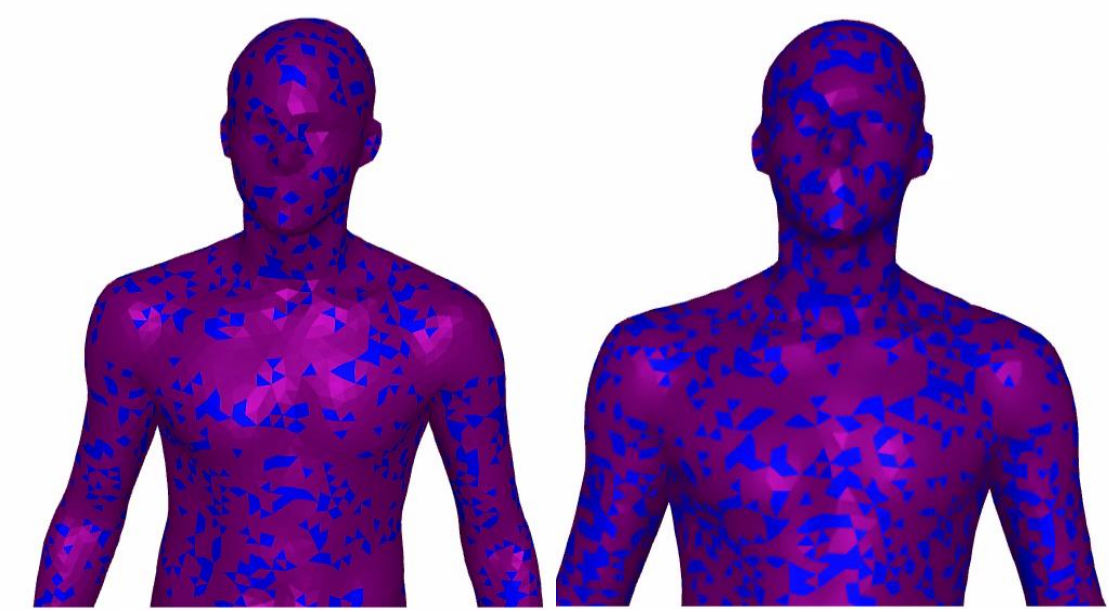


Figure 211 Progression from end of first 1/4 of the first epoch (left) to end the first epoch (right). Blue faces have an average error < 15`.

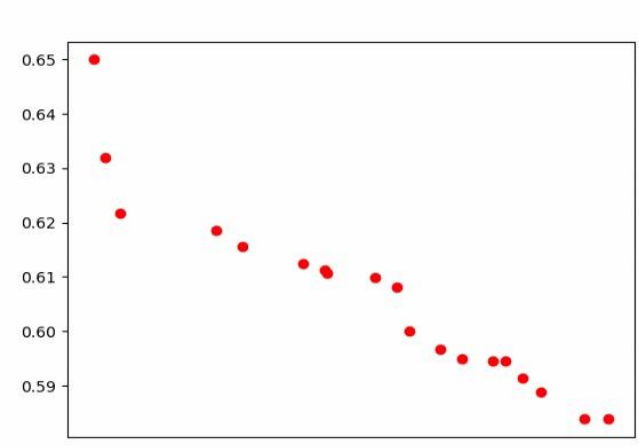


Figure 22 Loss progress on run using the principal directions

6.3 LIMITATIONS AND NOTICEABLE RESULTS

During this project the biggest hurdles found were based upon hardware, the use of a GPU noticeably improved the performance of the training time, the most common problem found when training was the lack of memory, justifying the need for smaller batches and compromising the ideal set up for training (the use of Implementations A and B). Another consequence of memory availability was the model size, as the increase of the model size greatly raised the use of memory and limited the size of batches, fortunately the results did not prove that the model sizes did only have a big impact on the results. Different types of models proved to have different constraints on memory. With the best single model result provided by DiffusionNet on its own.

Noticeably the marginal increase of batch sizes increases the speed of training on complex models but did not provide particularly better results.

The reduction of importance of the module did slow training as the optimization of the angles showed to be harder for the model, while the optimization of the module provided visible fast conversion from the start.

Notably the high resolution in meshes proved to slow down calculations noticeably besides the high requirements of memory they have based purely on the amount of data they require. Calculations for these “high resolution” meshes lead to slow calculations due to the number of calculations needed, as we had to a permutation on each face due to our custom loss function.

Our best results provided by DiffusionNet with principal directions as part of the feature vector, this implementation is noticeably good but has slow training time (approximately 3 weeks per epoch), and full training, consisting of 100 epochs, is beyond what is expected of this project.

7 CONCLUSION AND FUTURE RESEARCH

7.1 CONCLUSION

In this project we tried a hands-on approach to remeshing 3D models with the explicit objective of creating a first step for a parametric remesh algorithm. We focused particularly on human 3D models with a specific topology in mind. We curated a database of models and prepared them to assimilate our use case as closely as possible, and based on existing work, we designed a network capable of predicting a frame field using a face-based representation, composed of two mayor networks to capture global geometric data as well as local data.

The designed network was thoroughly tested to find the best implementation and confirm each choice made during the development. We confirmed the need to use a custom loss function, although it led to a big increase in training times. We confirmed that a global network with local data can provide an accurate enough estimation of the desired frame field to be used in parametric algorithms, but the elevated training times required for a full experiment are beyond what is in scope for this project. Yet the projection of the results given enough training time led to believe that this implementation is good enough to drive a remeshing algorithm.

Future research can follow up on the same model using more advanced setup and longer training times expecting the conversion of the training to yield better results.

Alternatively, other methods that were considered but discarded, that could prove interesting to explore included segmentation-based approaches and correspondence-based ones. Where the construction of a final mesh is left outside of the scope of the project and simple correspondence and transferring of vertices to correct positions can be done. This method is in line with proven direct use of the networks and could lead to promising results.

Finally, most networks do not work with high resolution meshes, since memory constraints or time constraints limit their capabilities (as can be seen in Table 1). A possibility that was not explored was the use of two meshes to do the predictions, one high resolution mesh to capture local data in detail and a coarse mesh with a limited number of vertices to allow for fast calculations on the global scale, this would be an approach like the one implied by the grouping done by PointNet++, yet the manual selection done to each mesh could lead to better results with greater control for level of detail on specific parts of each mesh.

8 BIBLIOGRAPHY

[1] K. Crane, "Carnegie Mellon University - brickisland," 16 april 2023. [Online]. Available: https://brickisland.net/ddg-web/#courseinfo. [Accessed november 2023].

[2] D. Kovacs, J. Bisceglio and D. Zorin, "Dyadic T-mesh subdivision," *ACM Trans. Graph.,* vol. 34, July 2015.

[3] N. Sharp and K. Crane, "You Can Find Geodesic Paths in Triangle Meshes by Just Flipping Edges," *ACM Trans. Graph.,* vol. 39, 2020.

[4] N. Villanueva, "Beginning 3D Game Assets Development Pipeline: Learn to Integrate from Maya to Unity", 2021.

[5] B. Tindall, The Art of Moving Points, Hippydrome Publishing, 2013.

[6] 3D-ace, "https://3d-ace.com/," 3 January 2024. [Online]. Available: https://3d-ace.com/blog/polygon-count-in-3d-modeling-for-game-assets/. [Accessed 2024].

[7] D. Khan, A. Plopski, Y. Fujimoto, M. Kanbara, G. Jabeen, Y. J. Zhang, X. Zhang and H. Kato, "Surface Remeshing: A Systematic Literature Review of Methods and Research Directions," *IEEE Transactions on Visualization and Computer Graphics,* vol. 28, pp. 1680-1713, 2022.

[8] M. Hussain, Y. Okada and K. Niijima, "A fast and memory-efficient method for LOD modeling of polygonal models," in *2003 International Conference on Geometric Modeling and Graphics, 2003. Proceedings*, 2003.

[9] M. Hussain, "Fast Decimation of Polygonal Models," 2008.

[10] K. Hu, D.-M. Yan, D. Bommes, P. Alliez and B. Benes, *Error-Bounded and Feature Preserving Surface Remeshing with Minimal Angle Improvement,* 2016.

[11] Y. Zhuang, M. Zou, N. Carr and T. Ju, "Anisotropic geodesics for live-wire mesh segmentation," *Computer Graphics Forum,* vol. 33, pp. 111-120, 2014.

[12] W. Lorensen and H. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *ACM SIGGRAPH Computer Graphics,* vol. 21, pp. 163-, August 1987.

[13] B. Delaunay, «Sur la sphère vide,» *Bulletin de l'Académie des Sciences de l'URSS. Classe des sciences mathématiques et na,* vol. 1934, p. 793–800, 1934.

[14] L. Chen and M. Holst, "Efficient mesh optimization schemes based on Optimal Delaunay Triangulations," *Computer Methods in Applied Mechanics and Engineering,* vol. 200, pp. 967-984, 2011.

[15] D. A. Field, "Laplacian smoothing and Delaunay triangulations," *Communications in Applied Numerical Methods,* vol. 4, pp. 709-712, 1988.

[16] J. Vollmer, R. Mencl and H. Muller, "Improved Laplacian Smoothing of Noisy Surface Meshes," *Computer Graphics Forum,* 1999.

[17] A. Nealen, T. Igarashi, O. Sorkine and M. Alexa, "Laplacian mesh optimization," in *Proceedings of the 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia*, New York, NY, USA, 2006.

[18] S. Lloyd, "Least squares quantization in PCM," *IEEE Transactions on Information Theory,* vol. 28, pp. 129-137, 1982.

[19] Y. Liu, W. Wang, B. Lévy, F. Sun, D.-M. Yan, L. Lu and C. Yang, "On Centroidal Voronoi Tessellation–Energy Smoothness and Fast Computation," *ACM Transactions on Graphics,* vol. 28, p. Article 101, August 2009.

[20] M. Audette, D. Rivière, M. Ewend, A. Enquobahrie and S. Valette, "Approach-guided controlled resolution brain meshing for FE-based interactive neurosurgery simulation," in *Workshop on Mesh Processing in Medical Image Analysis, in conjunction with MICCAI 2011.*, Toronto, 2011.

[21] S. Valette and J.-M. Chassery, "Approximated Centroidal Voronoi Diagrams for Uniform Polygonal Mesh Coarsening," *Computer Graphics Forum,* vol. 23, pp. 381-389, 2004.

[22] S. Valette, "https://github.com," 07 2024. [Online]. Available: https://github.com/valette/ACVD. [Accessed 2024].

[23] Q.-C. Xu, D.-M. Yan, W. Li and Y.-L. Yang, "Anisotropic Surface Remeshing without Obtuse Angles," *Computer Graphics Forum,* 2019.

[24] D. Zhang, F. He, S. Han, L. Zou, Y. Wu and Y. Chen, "An efficient approach to directly compute the exact Hausdorff distance for 3D point sets," *Integrated Computer-Aided Engineering,* vol. 24, pp. 261-277, June 2017.

[25] N. Ravi, J. Reizenstein, D. Novotny, T. Gordon, W.-Y. Lo, J. Johnson and G. Gkioxari, "Accelerating 3D Deep Learning with PyTorch3D," *arXiv:2007.08501,* 2020.

[26] J. Bush, M. W. J. Ferguson, T. Mason and G. McGrouther, "The dynamic rotation of langer's lines on facial expression," *Journal of Plastic, Reconstructive &Amp; Aesthetic Surgery,* vol. 60, no. 4, pp. 393-399, 2007.

[27] E. Catmull and J. Clark, "Recursively generated B-spline surfaces on arbitrary topological meshes," *Computer-Aided Design,* vol. 10, pp. 350-355, 1978.

[28] D. Bommes, B. Lévy, N. Pietroni, E. Puppo, C. Silva, M. Tarini and D. Zorin, "Quad-Mesh Generation and Processing: A Survey," *Computer Graphics Forum,* vol. 32, pp. 51-76, 2013.

[29] D. Panozzo, E. Puppo, M. Tarini and O. Sorkine-Hornung, "Frame Fields: Anisotropic and Non-Orthogonal Cross Fields," *ACM Transactions on Graphics (proceedings of ACM SIGGRAPH),* vol. 33, p. 134:1–134:11, 2014.

[30] J. Palacios and E. Zhang, "Rotational symmetry field design on surfaces," *ACM SIGGRAPH 2007 papers,* 2007.

[31] M. Fisher, P. Schröder, M. Desbrun and H. Hoppe, "Design of tangent vector fields," *ACM Trans. Graph.,* vol. 26, p. 56–es, July 2007.

[32] E. Zhang, K. Mischaikow and G. Turk, "Vector field design on surfaces," *ACM Trans. Graph.,* vol. 25, p. 1294–1326, October 2006.

[33] F. Knöppel, K. Crane, U. Pinkall and P. Schröder, "Globally optimal direction fields," *ACM Trans. Graph.,* vol. 32, July 2013.

[34] Dielen, Alexander and Lim, Isaak and Lyon, Max and Kobbelt, Leif, "Learning Direction Fields for Quad Mesh Generation," *Computer Graphics Forum,* vol. 40, no. 5, 2021.

[35] C. R. Qi, H. Su, K. Mo and L. J. Guibas, "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation," *arXiv preprint arXiv:1612.00593,* 2016.

[36] D. Guo, H. Liu, H. Zhao, Y. Cheng, Q. Song, Z. Gu, H. Zheng and B. Zheng, "Spiral Generative Network for Image Extrapolation," in *The European Conference on Computer Vision (ECCV)*, 2020.

[37] N. Girard, D. Smirnov, J. Solomon and Y. Tarabalka, "Polygonal Building Extraction by Frame Field Learning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.

[38] N. Pietroni, E. Puppo, G. Marcias, R. Scopigno and P. Cignoni, "Tracing Field-Coherent Quad Layouts," *Computer Graphics Forum,* vol. 35, pp. 485-496, 2016.

[39] A. Vaxman, M. Campen, D. Panozzo, O. Diamanti, K. Hildebrandt, D. Bommes and M. Ben-Chen., *Directional Field Synthesis, Design, and Processing,* Los Angeles: SIGGRAPH 2017, 2017.

[40] Z. Chen, *A Review of Deep Learning-Powered Mesh Reconstruction Methods,* 2023.

[41] N. Lei, Z. Li, Z. Xu, Y. Li and X. Gu, "What's the Situation With Intelligent Mesh Generation: A Survey and Perspectives," *IEEE Transactions on Visualization and Computer Graphics,* pp. 1-20, 2023.

[42] Y. Guo, H. Wang, Q. Hu, H. Liu, L. Liu and M. Bennamoun, *Deep Learning for 3D Point Clouds: A Survey,* 2020.

[43] A. Chang-Hoi, L. Sang-Soo, L. Hyuek-Jae and L. Soo-Young, "A self-organizing neural network approach for automatic mesh generation," *IEEE Transactions on Magnetics,* vol. 27, pp. 4201-4204, 1991.

[44] S. Yao, B. Yan, B. Chen and Y. Zeng, "An ANN-based element extraction method for automatic mesh generation," *Expert Systems with Applications,* vol. 29, pp. 193-206, 2005.

[45] X. Chen, T. Li, Q. Wan, X. He, C. Gong, Y. Pang and J. Liu, "MGNet: a novel differential mesh generation method based on unsupervised neural networks," *Engineering with Computers,* vol. 38, pp. 1-13, March 2022.

[46] D. Smirnov, M. Bessmeltsev and J. Solomon, "Deep Sketch-Based Modeling of Man-Made Shapes," *CoRR,* vol. abs/1906.12337, 2019.

[47] Z. Deng, Y. Liu, H. Pan, W. Jabi, J. Zhang and B. Deng, *Sketch2PQ: Freeform Planar Quadrilateral Mesh Design via a Single Sketch,* 2022.

[48] A. Dielen, I. Lim, M. Lyon and L. Kobbelt, "Learning Direction Fields for Quad Mesh Generation," *Computer Graphics Forum,* vol. 40, 2021.

[49] G. Yang, S. Belongie, B. Hariharan and V. Koltun, "Geometry Processing with Neural Fields," in *Thirty-Fifth Conference on Neural Information Processing Systems*, 2021.

[50] Y. Feng, Y. Feng, H. You, X. Zhao and Y. Gao, *MeshNet: Mesh Neural Network for 3D Shape Representation,* 2018.

[51] F. Bogo, J. Romero, G. Pons-Moll and M. J. Black, "Dynamic FAUST: Registering Human Bodies in Motion," in *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[52] M. Loper, N. Mahmood, J. Romero, G. Pons-Moll and M. J. Black, "SMPL: A Skinned Multi-Person Linear Model," *ACM Trans. Graphics (Proc. SIGGRAPH Asia),* vol. 34, p. 248:1–248:16, October 2015.

[53] N. Sharp, S. Attaiki, K. Crane and M. Ovsjanikov, *DiffusionNet: Discretization Agnostic Learning on Surfaces,* 2022.

[54] G. Qian, Y. Li, H. Peng, J. Mai, H. A. A. K. Hammoud, M. Elhoseiny and B. Ghanem, *PointNeXt: Revisiting PointNet++ with Improved Training and Scaling Strategies,* 2022.

[55] C. R. Qi, L. Yi, H. Su and L. J. Guibas, *PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space,* 2017.

[56] R. Hanocka, A. Hertz, N. Fish, R. Giryes, S. Fleishman and D. Cohen-Or, "MeshCNN: a network with an edge," *ACM Transactions on Graphics,* vol. 38, p. 1–12, July 2019.

[57] R. Wiersma, E. Eisemann and K. Hildebrandt, "CNNs on Surfaces using Rotation-Equivariant Features," *Transactions on Graphics,* vol. 39, July 2020.

[58] D. Smirnov and J. Solomon, *HodgeNet: Learning Spectral Geometry on Triangle Meshes,* 2021.

[59] A. Jacobson, D. Panozzo and others, *libigl: A simple C++ geometry processing library,* 2018.

[60] Y. Zhou, X. Cai, Q. Zhao, Z. Xiao and G. Xu, "Quadrilateral Mesh Generation Method Based on Convolutional Neural Network," *Information,* vol. 14, 2023.

[61] F. Williams, *Point Cloud Utils,* 2022.

[62] A. Vaxman and others, *Directional: A library for Directional Field Synthesis, Design, and Processing.*

[63] S. Valette, J. M. Chassery and R. Prost, "Generic Remeshing of 3D Triangular Meshes with Metric-Dependent Discrete Voronoi Diagrams," *IEEE Transactions on Visualization and Computer Graphics,* vol. 14, pp. 369-381, 2008.

[64] O. Topsakal, T. C. Akinci, J. Murphy, T.-J. Preston and M. Celikoyar, "Detecting Facial Landmarks on 3D Models Based on Geometric Properties – A Review of Algorithms, Enhancements, Additions and Open-Source Implementations," *IEEE Access,* vol. PP, pp. 1-1, January 2023.

[65] M. S. Smit and W. F. Bronsvoort, "Integration of Design and Analysis Models," *Computer-aided Design and Applications,* vol. 6, pp. 795-808, 2009.

[66] N. Ray, B. Vallet, W.-C. Li and B. Lévy, "N-Symmetry Direction Field Design," *ACM Transactions on Graphics,* vol. 27, p. Article 10, 2008.

[67] A. A. A. Osman, T. Bolkart and M. J. Black, "STAR: A Sparse Trained Articulated Human Body Regressor," in *European Conference on Computer Vision (ECCV)*, 2020.

[68] A. Myles, N. Pietroni, D. Kovacs and D. Zorin, "Feature-aligned T-meshes," *ACM Trans. Graph.,* vol. 29, July 2010.

[69] M. Lyon, M. Campen and L. Kobbelt, "Simpler Quad Layouts using Relaxed Singularities," *Computer Graphics Forum,* vol. 40, 2021.

[70] D. P. Kingma and J. Ba, *Adam: A Method for Stochastic Optimization,* 2017.

[71] S. Gong, L. Chen, M. Bronstein and S. Zafeiriou, "SpiralNet++: A Fast and Highly Efficient Mesh Convolution Operator," in *Proceedings of the IEEE International Conference on Computer Vision Workshops*, 2019.

[72] M. Garland and P. S. Heckbert, "Surface Simplification Using Quadric Error Metrics," in *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, USA, 1997.

[73] X. Gao, K. Wu and Z. Pan, "Low-Poly Mesh Generation for Building Models," in *Special Interest Group on Computer Graphics and Interactive Techniques Conference Proceedings*, New York, NY, USA, 2022.

[74] M. El Rhazi, A. ZARGHILI and A. Majda, "Automated Detection of Craniofacial Landmarks on a 3D Facial Mesh," 2020.

[75] K. Crane, M. Desbrun and P. Schröder, "Trivial Connections on Discrete Surfaces," *Computer Graphics Forum (SGP),* vol. 29, pp. 1525-1533, 2010.

[76] Z. Chen, Z. Pan, K. Wu, E. Vouga and X. Gao, "Robust Low-Poly Meshing for General 3D Models," *ACM Trans. Graph.,* vol. 42, July 2023.

[77] M. Campen, D. Bommes and L. Kobbelt, "Quantized Global Parametrization," *ACM Trans. Graph.,* vol. 34, November 2015.

[78] M. Botsch and L. Kobbelt, "A Remeshing Approach to Multiresolution Modeling," in *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, New York, NY, USA, 2004.

[79] D. P. Amir Vaxman, D. B. Olga Diamanti and M. Ben-Chen., *Directional Field Synthesis,* GitHub, 2021.

[80] S. Zhao, Mathematical Foundations of Reinforcement Learning, Springer Nature Press, 2024.

[81] M. M. Bronstein, J. Bruna, T. Cohen and P. Veličković, "Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges," *arXiv preprint arXiv:2104.13478,* 2021.

[82] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu and D. Amodei, *Scaling Laws for Neural Language Models,* 2020.

[83] P. J. H. Victor Guillemin, Differential Forms, (Draft version for MIT), 2018.

APPENDIX A:
SIMPLIFIED CATMULL-CLARK SUBDIVISION

Here we describe our simplified implementation of Catmull-Clark subdivision.

1. For every face F from the original Mesh M calculate its center VC, as the average of every vertex V of F.
   1.1. Save as a new vertex.
2. For every edge E, calculate the center point as the average of its vertices V_i V_j. These new vertices will be called $VE_{i,j}$
   2.1. Save the new vertices.
3. Create the new faces with V_i, VC, $VE_{i,i+1}$ and $VE_{i-1,i}$
   Keep in mind that we must preserve the face "sign", so the order of the vertices matters.
   To keep the same sign as the original F for every new face was registered as
   $[Vi, VE_{i,,i+1}, VC, VE_{i-1,i}]$
   3.1. Save the new faces and delete the old one.
4. Deleted all duplicate vertices created during the process (was faster than doing a search for each newly created vertex).
   In total every $VE_{i,j}$ should have been created twice for a closed surface.

This is a simplified application of Catmull-Clark subdivision where we don't calculate new position for the original vertices and the position of the new $VE_{i,j}$ are on the original edges. We prefer this application since we want the new mesh to have a geometry as close as possible to that of the original.

APPENDIX B: CUDA

CUDA is a platform for parallel computing and a model created by NVIDIA. Originally launched in 2006, CUDA is an API to allow developers to access the GPU and perform calculations on it. The CUDA API is an extension of the C programming language that gives access to the GPUs instruction set to developers, more commonly some preprogrammed libraries are already available to perform calculations using the GPU and take advantage of its increased capacity. Since 2015 NVIDIA has focused on CUDA for the development of neural networks.

The use of CUDA for training was fundamental as initial experiments proved that it improved throughput by a factor of x25.

APPENDIX C:
USE OF 1 MESH FOR EVALUATION AGAINST MULTIPLE MESHES

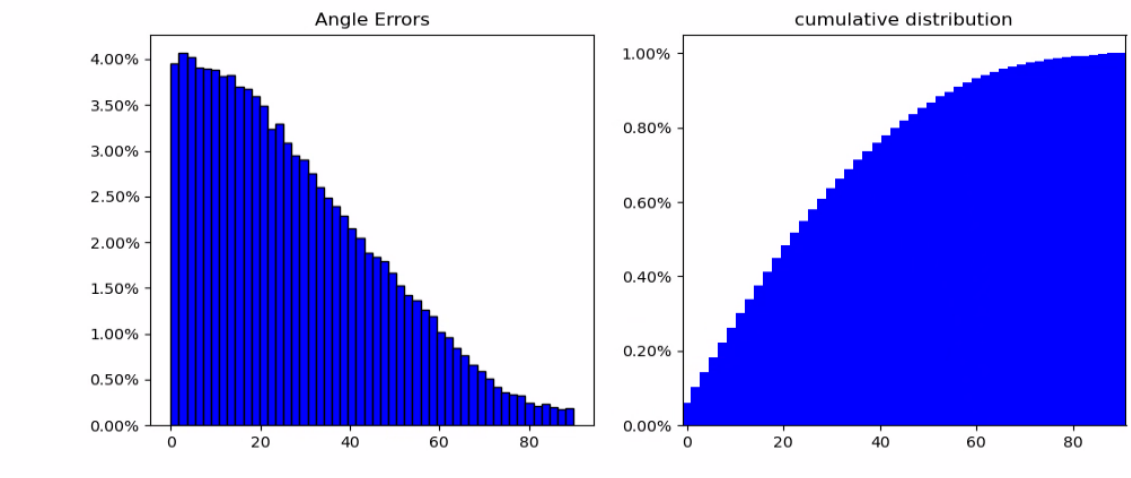
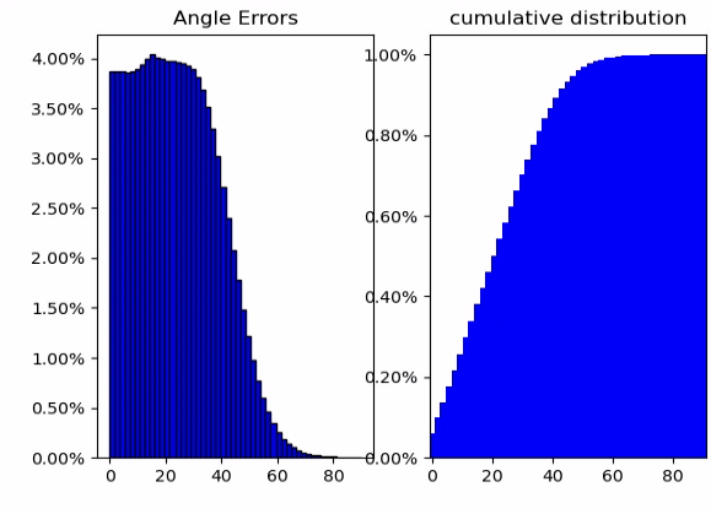


Figure 22 Result of angle errors on a single mesh from the testing set using PointNeXt.



In Figure 22 we can observe the errors of angles on a single ~60 k faced trimesh. Over 60% of faces have an error less than 40`. It is better to evaluate in a per mesh basis this error as the accumulation of “acceptable” errors leads to a steep graph, making it seem as if we have better results than we do, as can be seen in Figure 23

Figure 23 Angle errors cumulatively over 100 meshes