Learning Constrained Shape Spaces for Mesh Design

by

George Augusto Lorenzetti

A thesis submitted in partial fulfillment of the requirements for the degree of

Master in Game and Media Technology

Department of Information and Computing Sciences
Utrecht University

Abstract

Designing free-form structures in architecture is a difficult process, as constraints required for different building scenarios can be complex and typically require many design iterations involving multiple parties. Generating constrained three dimensional meshes through the use of neural networks provides an opportunity to simplify this process. In this paper we looked at generating constrained meshes using an autoencoder framework. Previous work had addressed methods for constraining free form quad meshes numerically, and more structured objects through the use of generative neural networks but generating free form constrained meshes has not been achieved thus far. In this work we present an autoencoder framework for generating quad meshes with constraints that fix vertices to specified points or planes. Results of mesh generation are limited to moderate, however emergent in our methodology is an additional contribution of creating an integration network that performs integrations converting quad meshes from edge length and dihedral angle representation to vertex coordinates. The performance of the integration network provides a number of benefits over numerical optimisation methods of integration, and also allows for smooth interpolation between meshes based on edge lengths and dihedral angles.

Table of Contents

Chapter 1

Introduction

Designing free-form structures in architecture is not a straight forward process. Structures are constrained in shape based on a number of factors such as structural integrity, panel manufacturability and building cost. For an architect, designing a free-form structure that adheres to construction constraints involves a costly and time consuming cycle of designing and fitting the design to constraints. Research into constraint spaces for polyhedral meshes has useful application in architecture, allowing for the development of tools which architects can use to explore shapes and designs directly in the defined constraint space of a mesh. The aim of such tools is ultimately to streamline the design process for architects, reducing the total cost and time investment required in designing free-form structures.

Previous work on constrained meshes for free-form architecture have involved computational models which are able to provide local approximations of a mesh's constraint space. Thus far, these methods have been computationally expensive and generally have large margins of error for large mesh deformations. Efforts have been made to reduce errors and achieve real-time computations, however these methods generally reduce the design freedom for the architect. Recently a number of works have used machine learning to accurately generate complex 3D objects such as furniture, and map soft body object deformations into a latent space using autoencoders. Thus far

however, machine learning has not been explored in generating constrained meshes for architecture.

The first contribution of our work is the creation of an integration network which can perform integrations converting meshes from edge length and dihedral angle form, to a list of vertex coordinates. Vertex coordinates are required when rendering a mesh but do not capture the geometry of a mesh as well as edge lengths and dihedral angles do. By working in edge length and dihedral angle form and having a network which integrates into vertex form, we are able to create an autoencoder that is better at capturing constraints, while still outputting meshes in a renderable form. On top of its importance in the problem of constraint generation, having an integration network also allows for interpolation between meshes based on edge lengths and dihedral angles. Interpolating using numerical optimisation algorithms is generally cumbersome, requiring re-optimisation at every intermediate step. By using an integration network for interpolation the process becomes much simpler, requiring only a single forward pass through the network at each step.

The second contribution we make provides only limited results, but involves creating a method for generating constrained meshes through the use of an autoencoder. The constraint scope of the work focuses on constraints that fix a point or set of points in a mesh to a specific plane or points. Part of this work involves using the integration network to convert an input mesh from a nonlinear representation of edge lengths and dihedral angles, to a list of vertex coordinates. The edge length and dihedral angle representation is used as it captures geometry associated with constraints such as planarity and the fixed point constraints that we focus on. Vertex representation is also needed for rendering the output meshes. A span of different quad meshes is encoded into the latent space of our autoencoder framework, and exploration of the constraint space is done by optimising within the latent space.

Chapter 2

Literature Review

2.1 Previous Work

2.1.1 Architectural Geometry

Research and Development in the field of architectural geometry aims to create tools which allow users to freely manipulate and explore the constraint space of architectural meshes. Early work in the field defined a mesh as a manifold with non-linear constraints represented as equality functions [Yan+11]. The constraint space was then defined as the intersection of the constraints in the manifold space. They then make a first order approximation of the constraint space using the manifold's tangent space, and a second order approximation using its oscilant. The constraint space can be explored by sampling along the oscilant, however direct manipulation and deformation of meshes is not supported. This work provided a solid foundation of research however it was computationally slow, meaning that sampling can not be done in real-time. On top of its computational time, the oscilant proves to be a useful estimate only for small deformation of the mesh, with larger deformations moving further away from the actual constraint space and increasing the degree of error.

In 2012 two works were produced, which aimed to improve application of the previous work as a design assistance tool [HK12; YPM12]. Yong et al. took the second order oscilant approximation and introduced the ability to directly deform meshes.

In this approach users define curves on the mesh as anchors and are then able to deform around these anchor curves. While successfully improving application, this work still suffers from the same slow computational time. Habbecke and Kobbelt took a different approach, providing a solver for co-linear mesh constraints using an iterative method inspired by inverse kinematics. The solver runs in real-time and enforces constraints exactly, however it is very limited in that it only supports co-linear constraints and provides only a single 'best' solution with no functionality for design space exploration.

Also in 2012, Vaxman [Vax12] took the mesh manifold and constraint representations from Yang et al's 2011 work and introduced a method of deforming, using per-face affine maps to preserve planarity constraints on a mesh perfectly. While this method provided contributions in both error reduction and application of design assistance tools, shape space exploration was still unavailable in real time. Error reduction was achieved regarding planarity only and errors remain comparable to the second order approximation for other constraints such as fairness and concyclity thresholds. Later in 2014, Vaxman [Vax14] also introduced a novel approach which works in the projected space for constraints, using multi-resolution mesh editing. While this method performed relatively well compared with the state of the art at the time and introduced a new approach to the problem, it is not robust to large meshes and has a limited ability to deal with applications that require explicit transformations of vectors.

Deng et al. [Den+13] built upon the first order tangent space approximation of a mesh's constraint space and introduced a numerical method for mesh optimisation that is able to preserve constraints for larger deformations. As with the previous efforts, this work also supports both manual deformation and shape space exploration, both of which work in real time. This work was a strong step forward, achieving a

combination of the benefits of previous works. In this achievement, two new goals for the state of the art were introduced- the ability to support inequality constraints, and the prevention of self intersections of meshes in the solution space. Further work was able to build upon this framework by introducing support for inequality constraints however this came at the expense of support for constraint space exploration [Tan+15].

In 2015, an improved numerical solver was implemented which introduced the concept of hard and soft constraints in the exploration of the constraint space [Den+15]. These types of constraints are analogous to equality and inequality constraints. This same year, another method was introduced which generates and deforms polyhedral meshes using topology maps [PCG15]. While these works were a marked step forward in their previous state of the art, they highlight that the state of the art had moved away from the goal of defining a global constrained shape space, towards numerical solvers and optimisation in order to improve computation time for local approximations.

A survey on the research field of architectural geometry was done in 2015 by Pottman, Eigensatz, Vaxman and Wallner [Pot+15]. This survey covered previous research on the topics of constrained meshes and design assistance tools, and noted a number of directions that research can take from the current state of the art. The survey noted that previous efforts have primarily focused on local approximations of a constraint space, leaving a gap in research into global approximations. This research intends to fill this gap, using machine learning to find a more global approximation of constrained shapes spaces.

2.1.2 Constrained Shape Spaces in 3D Modelling and Animation

A number of notable works involving constrained shape spaces have also come out of the research fields of 3D modelling and animation. Two works in particular [Bok+12; Sch+17] take an approach to constrained 3D CAD model design for engineering. In these works, design is less free-form than in architectural geometry examples, and constraints are defined as functions in the models and parameterised. This approach allows for perfect adherence to constraints and low computational cost. Exploration of the design space is done parametrically and for Bokeloh et al. is also done through directly altering constraint axes. While these approaches are effective for their specific applications, for architectural meshes a more general constraint space needs to be defined to allow for more free-form design.

Schultz et al. [Sch+14], von Radziewsky et al. [Rad+16] and Heeren et al. [Hee+16; Hee+18] all take an approach to soft body deformation which build constrained shape spaces by using a sample dataset and building a constraint space from samples and defined constraints. Schultz et al. use partial keyframes as constraints which denote information such as average vertex velocity at a point in time, while von Radziewsky et al. use elasticity and other deformation factors as constraints. Heeren et al. focus on a method of interpolating within the constraint space, first using splines [Hee+16] and subsequently using Principal Geodesic Analysis [Hee+18]. While each of these approaches are able to produce high quality deformations, they are not generalised approaches and each rely on having sample data for the specific models to be deformed. These approaches are analogous to the work in this paper in that they are able to construct a shape space using a sample dataset, however through the use machine learning we aim to be able to create a more general constraint space for meshes.

2.1.3 Machine Learning for Shape Spaces

A recently emerging avenue of research is the use of machine learning to find shape spaces for different types of 3D objects. Thus far no work has been done specifically on constrained shape spaces for architectural meshes, however there are a number of approaches with different applications that can provide a basis for our work. StructureNet [Mo+19] is a framework which has heavily inspired our work, and as such will be discussed in detail in the next section. StructureNet is based primarily on two previous works, GRASS by li et al. [Li+17] and PointNet by Qi et al. [Qi+17]. GRASS is a machine learning framework that uses a Generative Adversarial Network for generating shapes for structured objects such as furniture. StructureNet takes a different approach to the same problem, however is more effective than GRASS in its ability to handle large data sets. PointNet is a Deep Neural Network architecture that is able to classify the shape geometry of point clouds. This architecture is then adapted by StructureNet for the generation of point cloud geometry for individual object parts.

Luo et al. [Luo+18] and Fulton et al. [Ful+19] have done work in using machine learning to create physical deformations of 3D soft bodies. Both approaches were able to produce high quality deformations for soft bodies input into the architecture. Fulton et al's work is particularly relevant to our work as they were able to create deformations by encoding a latent space using an autoencoder.

Masci et al. [Mas+15] have also produced relevant work in the form of Geodesic Convolutional Neural Networks. In this work they were able to devise a way of performing convolutions on a 3D mesh, which are effective for applications such as invariant descriptors, shape correspondence and shape retrieval. While this is not directly related to constrained shape spaces, it provides a basis for working on meshes

with neural networks.

2.2 StructureNet

StructureNet [Mo+19] has served as the primary inspiration for the work in this project. It is a state of the art example of geometry generation using Variational Autoencoders, capable of producing high quality models using a discrete object structure with continuous part geometry. Models used in tests were generally different types of furniture, however StructureNet can be trained to generate models for any structured object. StructureNet is of particular interest as its approach to using VAEs to generate objects from the latent space can be applied to constrained architectural meshes. I have experimented with an implementation of StructureNet in order to gain a hands on understanding of the framework's applications and limitations, specifically focusing on what parts of the work may be applicable for this project.

2.2.1 Continuous Geometry Generation

One of the key requirements of generating constrained shape space meshes is that the VAE must have the ability to generate continuous geometry for the mesh. The StructureNet framework consists of two encoder decoder pairs- the first of which is used for continuous object part geometry generation while the second is used for structuring each generated part in a discrete hierarchy. The approach taken for the first encoder could be adapted for mesh generation, as there are certain implicit constraints associated with the shape of each different object part.

With respect to constrained shape space meshes, StructureNet is limited in that the framework contains geometry encoders and decoders only for bounding box and point cloud shape representation (Figure 2.1). A main goal for this research is to provide output in a form that can be used directly by architects and 3D artists. Neither of StructureNet's shape representations are suitable for this goal, as bounding boxes

cannot accurately represent surface meshes and point clouds cannot represent surface meshes in a way which can be easily manipulated by architects while designing. As such, a new way to encode and decode surface meshes in a more usable form such as a three dimensional model will need to be used.

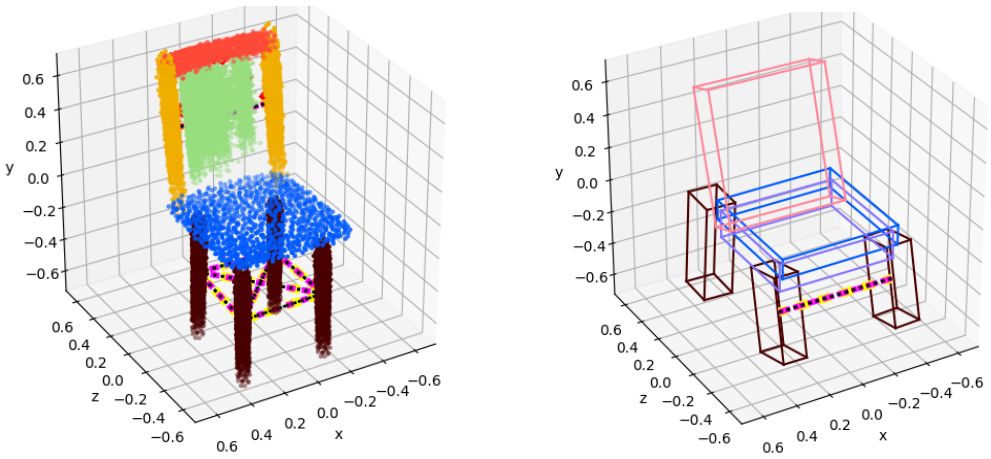


Figure 2.1: StructureNet supports two formats of object geometry representations. Bounding Box (left) and Point Cloud (right). Each representation uses a different geometry encoder and decoder with different architectures.

2.2.2 Latent Space Interpolation

The second point of interest that StructureNet presents in relation to our research is interpolation. One of the main goals in this project is to provide a framework which architects can use to experiment with shapes within the constraints of their design. An important part of this experimentation is the ability to interpolate between shapes such that intermediate results are also of high quality and remain within the constraint space. Interpolation within a learned latent space is an inherent advantage of using VAEs, however StuctureNet boasts more structurally aware interpolation between two objects within the latent space than previous related work. Intermediate results between interpolation in StructureNet are of particularly high quality, producing objects which would be feasible in real life (See Figure 2.2). Learning a latent space for

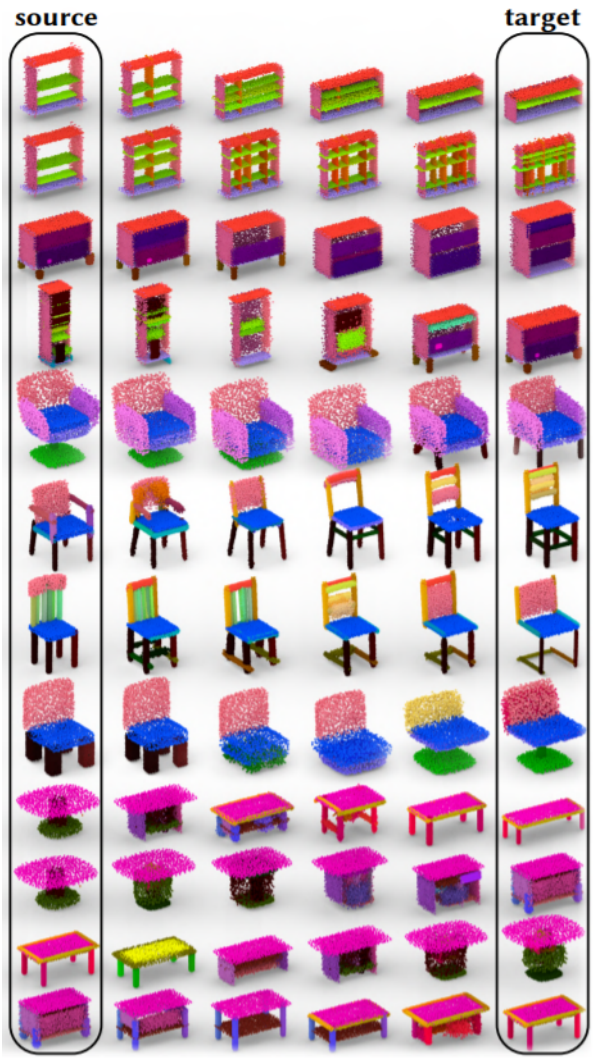


Figure 2.2: An example for each object type of interpolation within StructureNet's learned latent space. A number of high quality intermediate objects are generated while interpolating between the first and last outputs.

constrained shape meshes which can be interpolated in to produce a spectrum of high quality meshes, is an important goal for this project. While StructureNet does a good job with interpolation, it is limited in that it does not produce a continuous spectrum of high quality objects. Due to the nature of the structured objects that it can generate, interpolating between objects in the latent space produces many low quality, broken looking objects in between the intermediate high quality interpolations. Figure 2.3 shows two examples of these broken objects. For generating constrained shape space meshes and the intended applications, it is important to find a way to

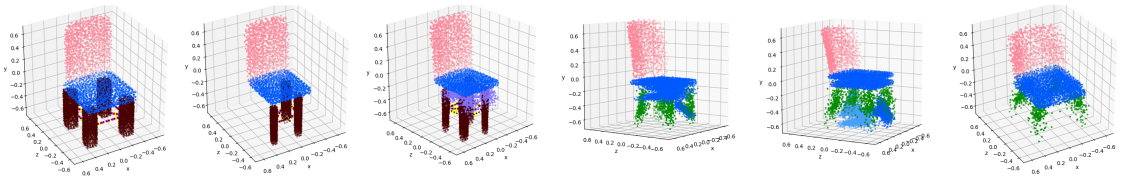


Figure 2.3: Interpolation between two chairs generated with random feature parameters. Broken or low quality intermediate generated objects were chosen to highlight how interpolating within the latent space does not produce a continuous spectrum of quality objects.

avoid such a limitation

2.2.3 Encoding Unannotated Images and Models

Another feature that StructureNet provides is the ability to take unannotated images, shapes or point cloud representations of objects and encode them into a common latent space. This is particularly interesting as an application for constrained shape space mesh generation as a feature such as this would allow architects and artists to provide mock up images or models as input to the system, and then be able to experiment with different shapes using the mock ups as a starting point. How useful StructureNet's method for this will be when applied to constrained architectural meshes is unclear though, as an object such as a chair is semantically very different to a continuous mesh surface. It is a promising concept however and is something the framework of this project can definitely benefit from.

2.2.4 Latent Space Dimensionality

A limitation of StructureNet with respect to this project is the dimensionality of the latent space. In generating constrained shape space meshes, this project aims to create a framework with which architects are able to explore different possible surface shapes within a set of given constraints. In order to effectively achieve this goal the number of features for each vector within the latent space should be relatively low,

and features should be semantically meaningful. StructureNet's learned latent space has 256 dimensions, which is too many for users to be able to experiment with in a meaningful way beyond trial and error. How applicable this limitation will be to this project is hard to predict however, as the output of StructureNet is very complex. The StructureNet framework requires the structural hierarchy of objects to be encoded, as well as the specific geometry of each part in the object. Output for generated constrained shape space meshes is likely to be less complex and as such could be effectively encoded in a latent space with much lower dimensionality.

Chapter 3

Background

3.1 Constraint Spaces

Constraints on physical objects are ubiquitous and appear in many forms- Movement of a pendulum is constrained along a semicircle of fixed radius, machinery socket joints are constrained in their degrees of freedom and tools such as wrenches are constrained in how they must be shaped. In free-form architecture, curved surfaces are modelled as meshes which have various constraints such as vertex position, face planarity, shape regularity, and size. Constraints are typically defined as functions of the form $C(x) = 0$, where x is the input object or system, and the constraint is adhered to perfectly if and only if the result is zero.

In this work, we focus on fixed point constraints on quad meshes. This is a simple starting point for generating constrained meshes, but implementation is done with the possibility in mind to extend the framework to different types of constraints. A quad mesh can formally be described as a set of vertices and faces of degree 4. Sets of meshes that share an identical connectivity defined as just their set of vertices $v \in \mathbb{R}^3$. To define a shape space for a set of meshes with shared connectivity, a mesh is modelled as a manifold $p \in \mathbb{R}^D$ where $p = \{v_0, v_1, v_2...v_n\}$ consisting of each of the n vertices in the mesh, and $D = 3n$. When imposing constraints on quad meshes to form a constrained shape space M, the set of meshes that lie within the intersection

of all m constraints $E_i(p) = 0$, $i = 1...m$ form the constrained shape space. This intersection of m constraints in $\mathbb{R}^D$ is formally defined as $\Gamma_i = \{\, p \in \mathbb{R}^D \mid E_i(p) = 0 \,\}$, $i = 1...m$. This work uses an autoencoder framework to encode Γ_i into a latent space that can be explored during the process of designing free-form architectural surfaces.

3.2 Autoencoders

Autoencoders are a type of neural network architecture that, once trained on a specific data set, are able to generate new examples of similar data. Autoencoders consist of two separate neural networks, an encoder and a decoder. The function of the encoder is to take a data sample as an input vector and encode it as a feature vector in a lower dimension space called the latent space. The decoder has the opposite function, it takes a sample from the latent space as input and outputs the corresponding data sample. Since the goal of an autoencoder is to encode data into a latent space and accurately decode samples from the latent space, error is measured by how accurately a sample can be encoded and then reconstructed with the decoder. If the functions $e(x)$ and $d(x)$ represent the encoder and decoder respectively, then the error of an autoencoder is measured by accuracy of the approximation $x \approx d(e(x))$ where x is an input sample of training data (Figure 3.1).

We take advantage of the generative capabilities of autoencoders to generate our constrained meshes. We train an autoencoder that encodes a latent space of deformed meshes with consistent size and connectivity, and generate meshes from the latent space which adhere to our prescribed constraints. Autoencoders however, are limited in their generative ability, since the distribution of their latent space reflects the distribution of the sample data that they are trained on. In the likely scenario that the training dataset is not distributed regularly, the latent space will not be regularly distributed and as such, sampling and decoding random points within the latent space is not likely to generate good quality new data.

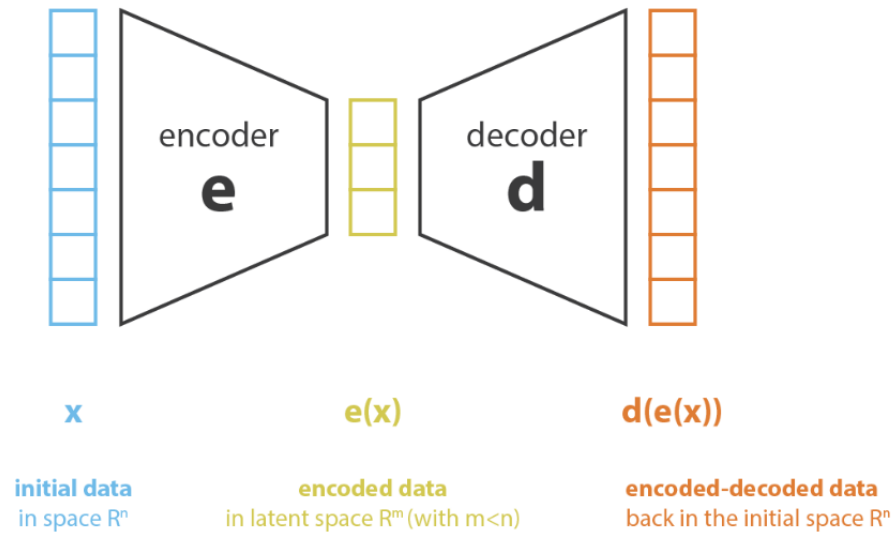


Figure 3.1: Autoencoder architecture consisting of two networks, the encoder and the decoder.[Roc19]

There are a number of ways around this, so that sampling can be done of the latent space with an autoencoder. The option that we choose is through optimisations. If the latent space is sufficiently smooth, sampling can be done by interpolating between known encoded latent vectors. This interpolation can just be a linear interpolation or in the case of generating constrained meshes, interpolating with an optimisation algorithm can be done to find meshes which fit constrainst. In the case of latent space optimisation, an initial guess and a cost function are provided to the optimizer. The cost function defines the desired constraints and the optimizer will iteratively converge onto a solution in the latent space which best fits the constraints.

Chapter 4

Methodology

In this chapter will cover the implementation details for how the final mesh generation framework was implemented. Details and justification of all stages of development from training data creation all the way up to actual mesh generation are covered.

4.1 Creating Training Data

Since the method for generating constrained meshes was to be done using an autoencoder, the first consideration that needed to be made was what the data set for network training should look like. For the sake of simplicity, we decided that to begin with, the focus should be on generating meshes with constant connectivity. As such, a data set for training the network should consist of different deformations of a single starting mesh. With this approach, the generated meshes would all be deformations of the starting mesh, generated specifically to adhere to the prescribed constraints.

The starting point for each training data set was a single quad mesh in the shape of a square plane. We then planned to create different data sets consisting of different sized meshes, used for different sets of experiments. Each mesh in the training data set was deformed using a set of eigenvectors calculated using the laplacian of

the starting mesh. This method was chosen because we wanted our autoencoder to generate smooth shapes that could feasibly be used in architecture. This eigenvector method of deformation produces smooth deformations, and hence a network trained on these meshes should retain this property when generating new meshes.

4.1.1 Calculating the Laplacian

The method described here, used for calculating the laplacian of the starting mesh was introduced by Alexa and Wardetzky [AW11], and is outlined in more detail in their paper. The laplacian of a mesh consists of two parts, a stiffness matrix W and a mass matrix M, both of dimension $v \times v$, where v is the number of vertices in the mesh. W and M are then used to solve the system $W\phi = M\phi\lambda$, to produce $v \times l$ matrix ϕ which consists of l eigenvectors as its columns, and a $l \times l$ diagonal matrix λ of corresponding eigenvalues.

The stiffness matrix W takes the form $W = d^T M_1 d$. Matrix d is a $(2e_i + e_b) \times (2e_i + e_b)$ coboundary matrix, where e_i and e_b are the respective numbers of inner and boundary edges in the mesh. Matrix M_1 is a $(2e_i + e_b) \times v$ inner product matrix. In order to calculate the laplacian and generate ϕ for mesh deforming, matrices M, d and M_1 must be assembled. Matrix M is a diagonal matrix where each value M_{pp} contains a mass value for vertex p and is calculated as follows:

$$M_{pp} = \sum_{f \ni p} \frac{|f|}{k_f} \tag{4.1}$$

Where for each face f adjacent to vertex p, $|f|$ is the largest signed area of projections of f on to all orthogonal planes in $\mathbb{R}^3$. An intuitive illustration for $|f|$ is shown in Figure 4.1.

Matrix d is a sparse coboundary matrix indicating each edge in the mesh. Inner edges are counted twice, once for each face they are a part of. Each entry d_{ij} corresponding to a directed edge e_{ij} takes the value $d_{ij} = \pm 1$ depending on the sign of the edge. Constructing matrix M_1 is more involved than M and d, and is outlined in the algorithm below. In the algorithm, parameter λ takes a chosen value, for our implementation $\lambda = 2$.

Algorithm 1 Assembling M_1 Input:Polygonal face f, parameter λ, matrix d

for each face f in mesh **do**
  $B, E, \bar{E} \in \mathbb{R}^{f \times 3}$
  for each vertex x_i in f **do**
    $E(i) = (x_{i+1} - x_i)^T$
    $B(i) = \frac{1}{2}(x_{i+1} + x_i)^T$
  end for
  $A = E^T B$
  $\tilde{M} = \frac{\sqrt{2}}{||A||} B B^T$
  $\bar{n} = normalized(-A_{23}, A_{13}, -A_{12})^T$
  for each vertex x_i in f **do**
    $\bar{x_i} = x_i - (x_i \cdot \bar{n})\bar{n}$
  end for
  for each vertex x_i in f **do**
    $\bar{E}(i) = (\bar{x_{i+1}} - \bar{x_i})^T$
  end for
  $C =$ orthonormal kernel of $\bar{E}^T$ (using LU then SVD)
  $U = \lambda Id$
  $M_f = \tilde{M} + CUC^T$
end for
$M_1 =$ diagonal matrix where each M_f is assembled along the diagonal in position of its corresponding face

4.1.2 Deforming Using Eigenvectors

After calculating the laplacian for the starting mesh, we obtained the eigenvectors ϕ by solving the system $W\phi = M\phi\lambda$ using MATLAB's *eigs* function. This function takes W, M and the number of eigenvectors l to return. In creating our data sets we chose to use 100 eigenvectors or, if the mesh size for the data set was too small for *eigs* to produce 100 then we used the maximum number possible. Deformations of

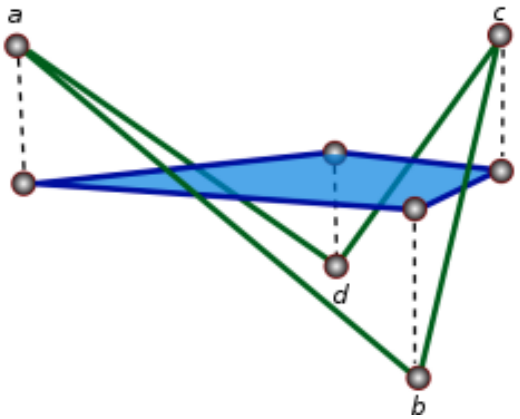


Figure 4.1: An illustration of the maximal vector area $|f|$ corresponding to face f_{abcd}. Illustration created by Alexa and Wardetzky [AW11]

the starting mesh were created by using linear combinations of the eigenvectors. Each combination was created with random coefficients, but weighted to bias eigenvectors corresponding to higher magnitude eigenvalues. As each eigenvector is of length v, three linear combinations were created for each deformation, used to deform each vertex's corresponding x, y and z coordinates. Figure 4.2 gives a visual on the result of deforming using this method.

4.2 Mesh Representation

4.2.1 Edge Lengths and Dihedral Angles

The most common way to represent a three dimensional mesh in a data set is through a combination of vertex coordinates and connectivity information. This representation is required for most graphical applications when visualising meshes and hence, this representation was the starting point we used for experiments. We did some initial experimentation in encoding three dimensional quad meshes into a latent space, using vertex coordinates as network input. Input vectors took the form $[x_1, y_1, z_1, x_2, y_2, z_2 \dots x_v, y_v, z_v]$ for a mesh containing v vertices, with connectivity data provided as needed for loss calculation but not as network input. This initial experimentation produced poor results and indicated that a more geometrically aware

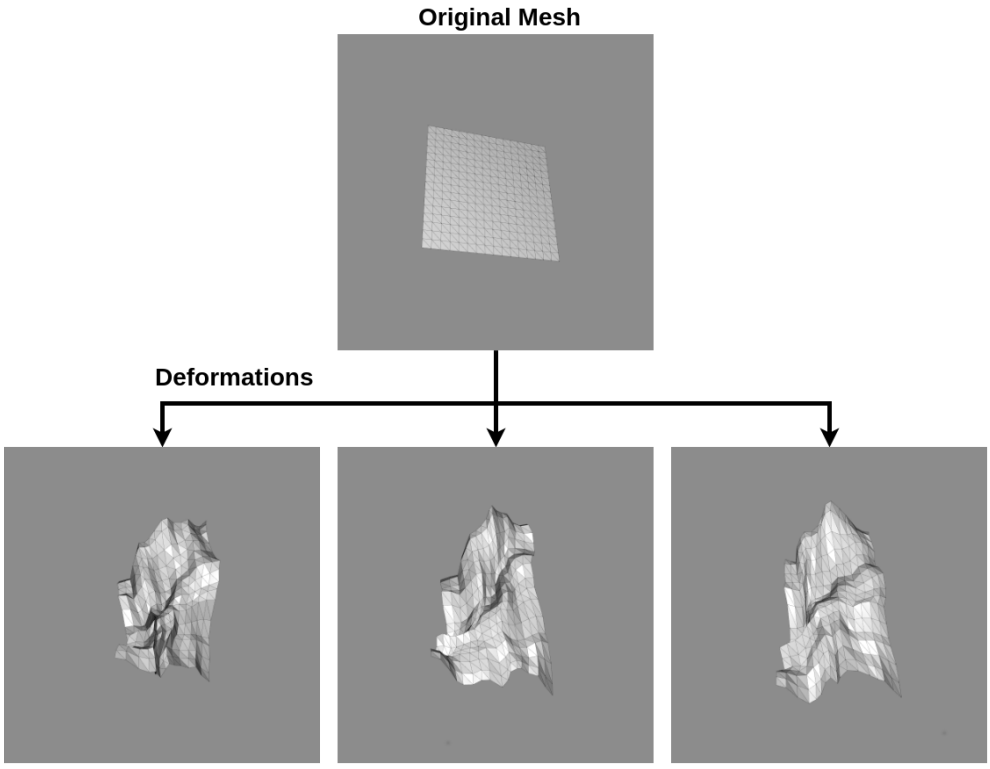


Figure 4.2: Three examples of meshes deformed using the eigenvector method.

mesh representation was needed.

Frölich and Bostch [FB11] present a different way to represent a three dimensional mesh, which they used for their work in generating soft-body mesh deformations. They represent a mesh by its edge lengths and dihedral angles- measuring the stretching and bending of a mesh as deviations in these values. Based on Frölich and Bostch's success in deforming meshes using this representation and the fact that our data sets contain meshes with a constant connectivity, we hypothesize that using edge lengths and dihedral angles as input will allow an autoencoder to better capture the geometry of a mesh.

When using edge length and dihedral angle representation, we include the lengths of each edge between two vertices, the diagonals of each face, the long edges between neighbouring neighbouring faces and the diagonals between neighbouring faces in the

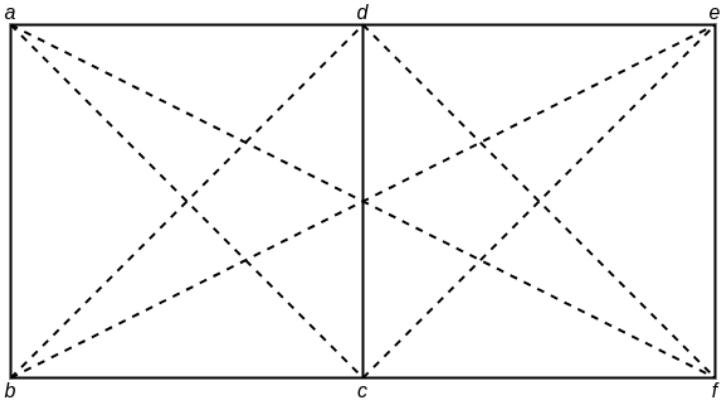


Figure 4.3: Given two neighbouring quads Q_{abcd} and Q_{cdef} in a mesh, our representation would include as edge lengths: edges $|\bar{ab}|$, $|\bar{bc}|$, $|\bar{cd}|$, $|\bar{da}|$, $|\bar{de}|$, $|\bar{ef}|$ $|\bar{fc}|$, face diagonals $|\bar{ac}|$, $|\bar{bd}|$, $|\bar{df}|$ $|\bar{ce}|$, long edges between neighbours $|\bar{ae}|$ $|\bar{bf}|$, and neighbour diagonals $|\bar{af}|$ $|\bar{be}|$.

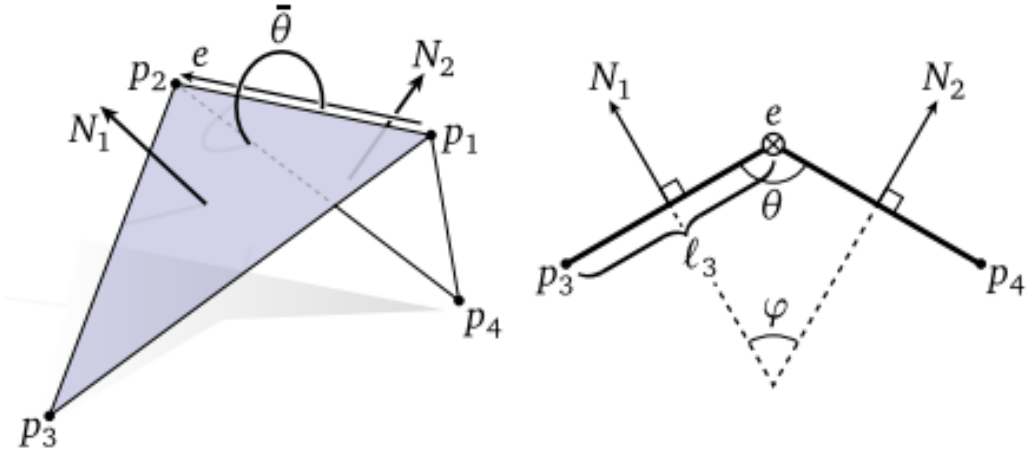


Figure 4.4: For the quad $Q_{p_1p_2p_3p_4}$, we calculate the dihedral angle θ. $N1$ and $N2$ are normals to the triangles $T_{p_1p_2p_3}$ and $T_{p_1p_2p_4}$ respectively. [Cra18]

mesh. Each of these edge lengths is illustrated in Figure 4.3. We calculate dihedral angles using a method outlined by Crane [Cra18]. For a quad shown in Figure 4.4, the dihedral angle θ is calculated using:

$$\theta = atan2(e \cdot (N1 \times N2), N1 \cdot N2) \tag{4.2}$$

Using this equation we calculate the dihedral angle for each face diagonal and each neighbour diagonal in the mesh. Input for our autoencoder is formatted as a list of all lengths, concatenated with a list of dihedral angles.

4.2.2 Consequences for Data Generation

Changing to representing training meshes as edge lengths and dihedral angles does provide an issue in mesh generation that we address in our data generation process. Edge lengths and dihedral angles do not provide any positioning information about the mesh, and we anticipated that this could cause problems in preserving these properties when generating meshes, and converting to vertex form. In order to preserve these properties, we add two extra steps to data generation. The idea behind these steps is to create consistency between training meshes, that will be reflected in network output.

The first step that we take is to address the problem of orientation. We do this by performing PCA on the deformed mesh before converting from vertex form. From this PCA we take the three principal component axes and rotate the mesh such that they align with the x, y and z Cartesian axes. This is done for each mesh in the training data sets, and provides a consistency in orientation that is reflected in the neural network output. The second step provides a similar consistency for translation. This step involves translating each training mesh such that the average position of all its vertices lies on the origin. After these steps are completed, the data is converted to edge length and dihedral angle form and the data generation is complete.

4.3 Integration Network

Despite our shift from representing meshes by their vertex coordinates to edge lengths and dihedral angles, we must still have a way to convert back to vertex coordinates so that generated meshes can be rendered. Calculating edge lengths and dihedral angles from a list of vertex coordinates with connectivity data is trivial, however converting back to vertex coordinates is not so simple. Converting from edge lengths and dihedral angles to vertex coordinates is an integration problem that would typically be

solved numerically using a non-linear least squares optimisation. Solving the integration numerically like this is too slow for an architecture design application however, because the non linear least squares algorithm would need to be run each time a user edits a mesh.

The need for a faster way to convert from edge lengths and dihedral angles, to vertex coordinates gives rise to a secondary goal in our work. This secondary goal is to train a neural network to solve the integration problem, instead of doing so numerically. This is beneficial because converting to vertex coordinates using a single forward pass through a network will be much faster than a numeric optimisation, and using a network will also allow the conversion to be built directly into the autoencoder framework. With this we present our integration network architecture, described in Figure 4.5. The integration network is a fully connected multi-layer perceptron consisting of 8 hidden layers, each with dimension 128, using the ELU activation function. The network takes the previously explained concatenated list of edge lengths and dihedral angles as input, and outputs a list of vertex coordinates. The size of input and output vectors for the integration network vary depending on the data set used for training, as larger meshes will have larger input and output vectors.

4.4 Generating Constrained Meshes

To generate constrained three dimensional meshes, we use an autoencoder framework. The architecture of the framework is shown in Figure 4.6. The decoder and encoder both consist of 5 hidden layers of dimension 512, with a latent space of dimension 256. The latent space dimension remains constant regardless of the input size. The autoencoder uses the exact same architecture for the integration network as previously described. Empirically, the autoencoder is able to encode a smooth latent space of three dimensional meshes. This allows for linear interpolation between two encoded meshes to generate meshes of consistent quality.

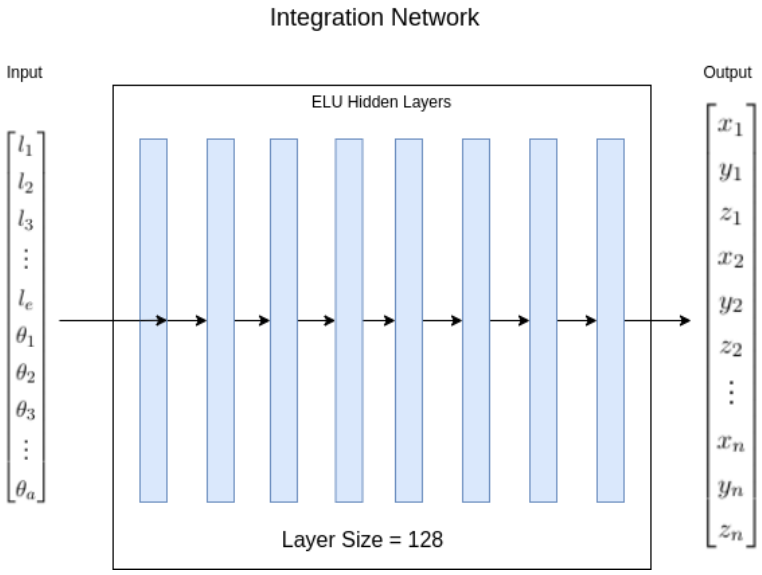


Figure 4.5: Architecture diagram of the integration network. For a mesh with n vertices, the network takes the list of edge lengths and dihedral angles as input and outputs the vertex positions of the mesh.

The scope of constraints that we focus on are constraints which fix a vertex or group of vertices to a specific point, set of points or plane. In order to generate meshes which adhere to these constraints, we utilise latent space optimisation to search the latent space for meshes which fit the constraints. This optimisation is possible because of the smooth latent space that our autoenncoder produces. For this optimisation we use the Limited-Memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) algorithm, which iteratively minimizes the following cost functions:

$$C_{point} = |l - l_0|^2 + w(\sum_{v \in F} |v - v_0|^2) \tag{4.3}$$

$$C_{plane} = |l - l_0|^2 + w(\sum_{v \in F} |v_p - v_{0p}|^2) \tag{4.4}$$

Depending on whether vertices are being constrained to a point or a plane, either C_{point} or C_{plane} is used as a cost function. In these functions, latent vector l is the current guess for the output and is updated each iteration, latent vector l_0 is the initial guess provided to the algorithm, F is the set of vertices that are being constrained, and v_0 is the fixed point corresponding to each vertex v. When fixing to only a plane,

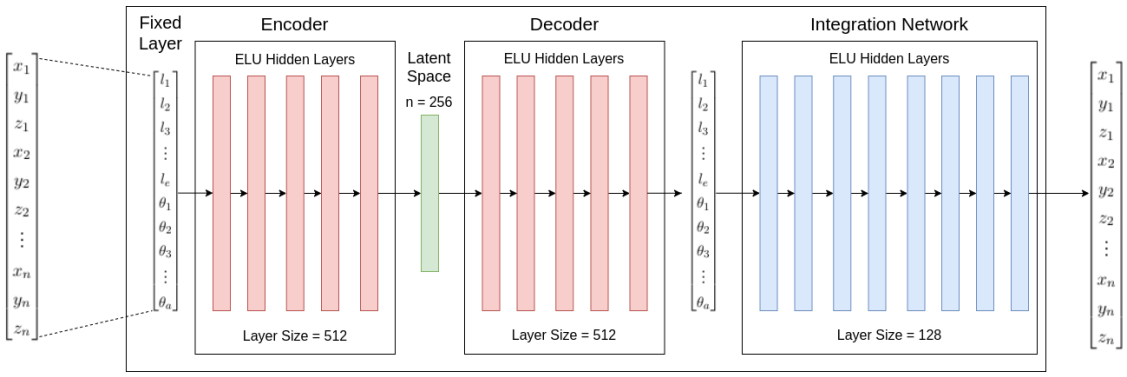


Figure 4.6: Autoencoder architecture.

v_p is the x, y or z coordinate of v, depending on which plane is being fixed. To provide a good quality initial guess to the L-BFGS, an example mesh from validation data is encoded using the autoencoder framework. The encoded latent variable is then provided to the optimisation algorithm as an initial guess for the interactive process.

Chapter 5

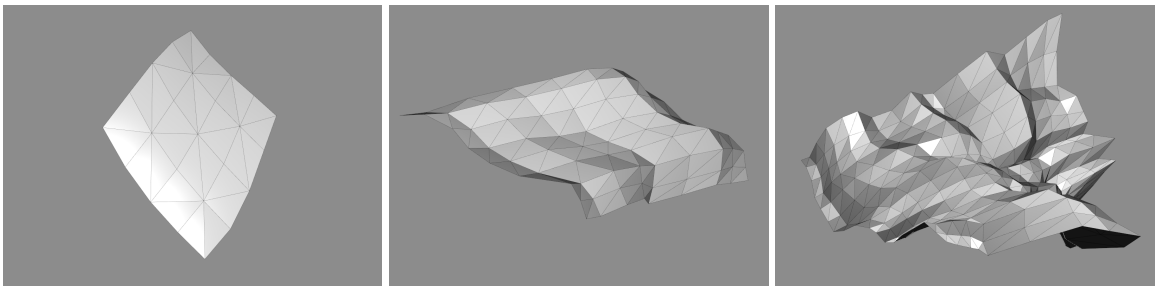
Experimentation

We split experiments into two separate sections- integration network, constrained mesh generation. The first section looks at the performance of the integration network in converting from the edge length and dihedral angle representation to vertex coordinate representation. The performance of the integration network will be measured in comparison with performance of a conventional non-linear least squares solution for the integration. The second section then looks at how well the autoencoder performs in generating new meshes that constrain certain vertices to fixed points and planes.

5.1 Integration Network

The first section of experiments focuses on the performance of the integration network. Experiments will be performed three times, each time using an integration network trained on a different data set. Each data set contains meshes of a different size (4x4, 8x8 and 16x16). Each data set is split into 5800 training meshes and 200 validation meshes, for a total size of 6000 meshes. All meshes in a data set are deformations of a single original mesh using the eigenvector deformation method described in the previous chapter. Examples of training data can be seen in Figure 5.1.

For each dataset, training parameters were kept constant and are outlined as follows: 500 training epochs, batch size of 800, learning rate of 1e-3, Mean Squared



(a) 4x4 (b) 8x8 (c) 16x16

Figure 5.1: An deformed mesh example from each data set. Meshes are 4x4, 8x8 and 16x16 quads in size.

Error (Pytorch implementation) as a loss function and Adam as the optimizer.

The goal of the integration network is to convert a mesh from our edge length and dihedral angle representation into a vertex coordinate representation. As this task would typically be solved using a numerically we compared output from a non-linear least squares algorithm with output from the network. A set of 200 validation meshes was converted to vertex coordinate form using a MATLAB implementation of the Levenberg-Marquardt algorithm, and then converted again using the integration network. For each reconstruction, the mean squared error between the vertex positions in the original mesh and reconstruction was calculated and the mean of these 200 values was used to compare the integration network with Levenberg-Marquardt.

In addition to experiments measuring the basic performance of the integration network, we perform experiments to assess linear interpolation between two meshes in edge length and dihedral angle format. Linear interpolation is identified as an additional application of the integration network as it allows for interpolation without having to re-optimise at each step, as is needed when interpolating numerically. To examine how well linear interpolation works, we interpolate between two validation meshes for each data set. We visualise each intermediate result and from this assess the quality of the interpolation as a whole.

5.2 Constrained Mesh Generation

The second set of experiments looked at how well constrained meshes could be generated by training the autoencoder and then performing optimisation in the latent space. As mentioned in the previous chapters, the types of constraints that we focus on are constraints which fix a single vertex or set of vertices or some fixed points or plane. The autoencoder was trained using the same three datasets that were used for the first set of experiments. For each data set, the same training parameters were used which are as follows: 400 training epochs, batch size of 1000, learning rate of 1e-3, Mean Squared Error (Pytorch implementation) as a loss function and Adam as the optimizer.

Using the trained autoencoder, we generate constrained meshes using the method outlined in chapter 4.4. We perform experiments for fixing vertices to both points and planes, and use the corresponding cost functions (4.3) and (4.4) in the latent space optimisations. To examine the robustness of the constrained mesh generation, different examples were generated with different constraints involving a single fixed vertex, multiple fixed vertices or a row of vertices fixed to a single plane. All three scenarios were tested for each of the three mesh sizes. To gauge how well vertices were fixed, a visualisation was used which shows each vertex's actual position in comparison with its desired constrained position.

For the three constraint variations we tested, generation was done via latent space optimisation, which requires a valid latent vector as an initial guess. To do this, a single validation example was encoded and used as an initial guess.

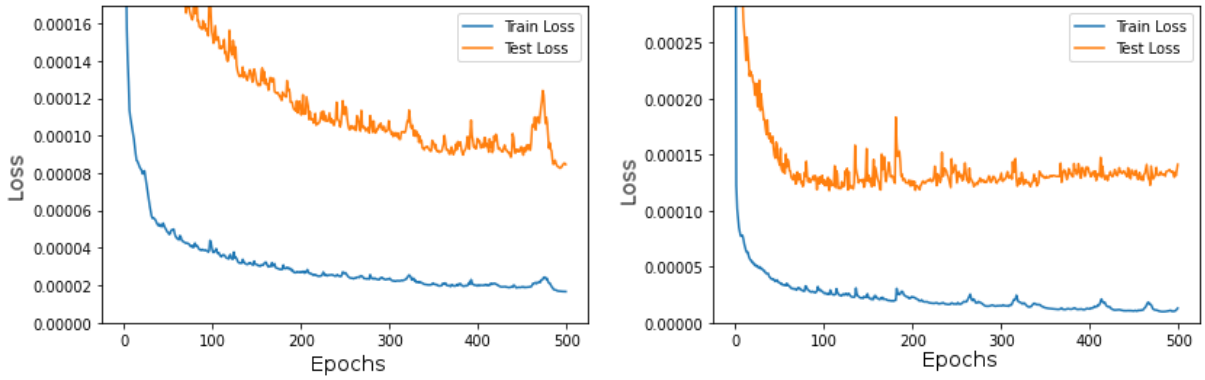
Chapter 6

Results

This section contains the results generated from the framework outlined in the previous section. Results are split into two separate sections for the two sets of experiments; the integration network, and generation of constrained meshes. All experiments in both sections were conducted on three different data sets, containing meshes of size 4x4, 8x8 and 16x16 quads.

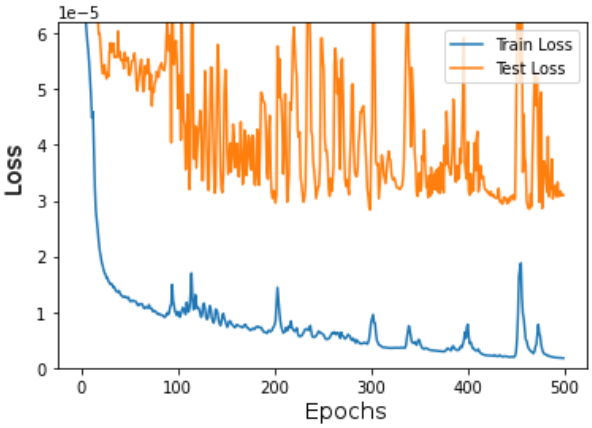
6.1 Integration Network

First we will look at the performance of the integration network. Experimental were performed as outline in chapter 5.1. As a benchmark for comparison, all experiments were conducted using integration network and also the Levenberg-Marquardt algorithm in MATLAB.



(a) 16x16

(b) 8x8



(c) 4x4

Figure 6.1: Losses recorded while training the integration network on each data set. Data sets consisted of 6000 meshes each, split into 5800 training samples and 200 test samples. Loss was measured using PyTorch's mean squared error loss function, comparing output vertex positions with a ground truth data set. The integration network training results in the best loss value on the 4x4 data set and worst on the 8x8 data set.

Error Per Data Set			
	4x4	8x8	16x16
Integration Network	0.018586	0.084803	0.050732
Levenberg-Marquardt	0.078949	0.155008	1.001953

Figure 6.2: The above table shows the error value for a set of 200 validation meshes for each mesh size. Error calculation was done as outlined in the previous chapter. The scores in the table show that the integration network performed significantly better than Levenberg-Marquardt for every mesh size.

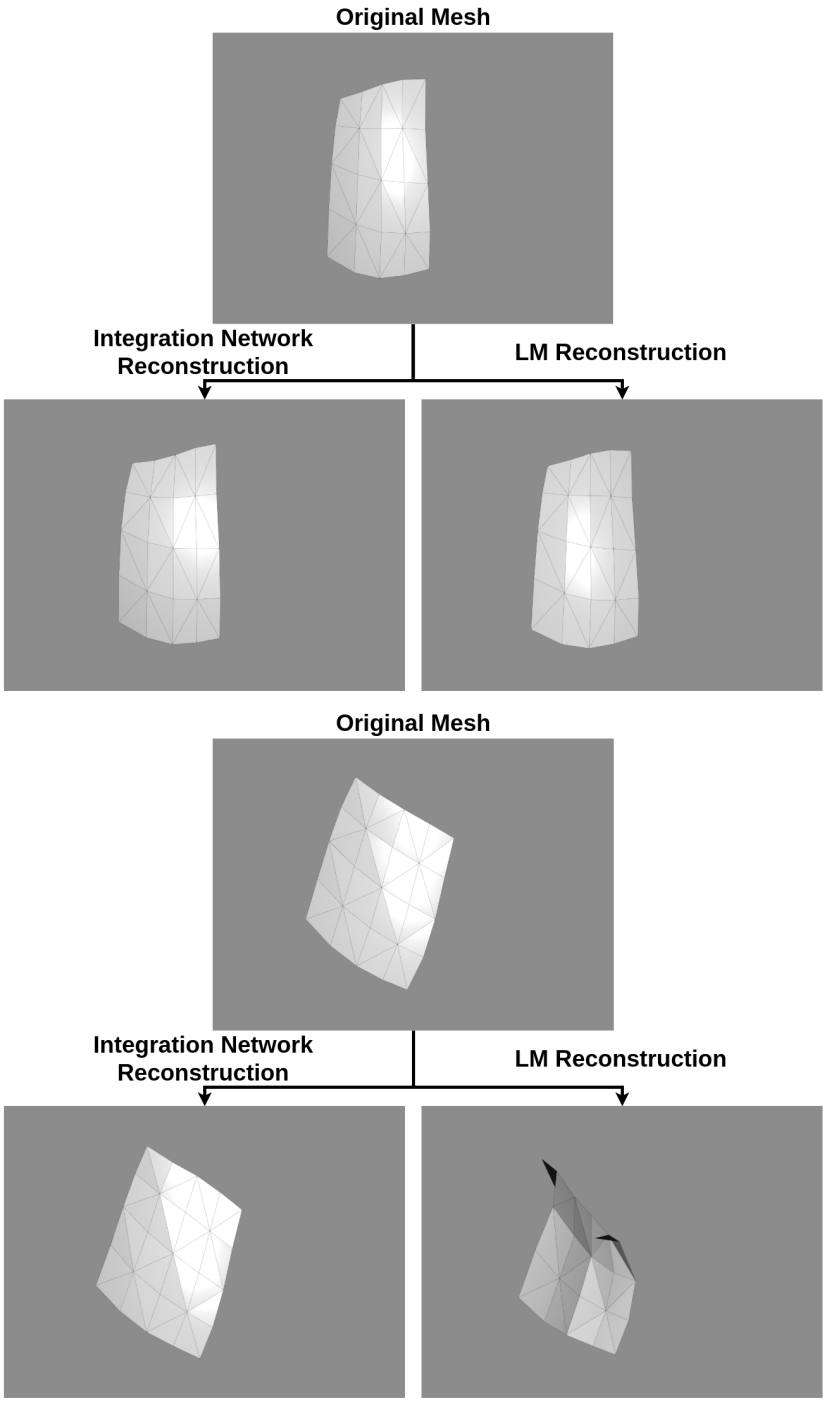


Figure 6.3: Two examples comparing the integration network with Leverberg-Marquardt output for 4x4 meshes. These examples were chosen to illustrate an observation made about the results. The Leverberg-Marquardt produces output with varying quality compared with the integration network. For some examples, Leverberg-Marquardt creates an almost perfect reconstruction while other examples are very poorly reconstructed. The integration network produces good, but less perfect reconstructions without suffering from the very poor outliers. These outliers could be attributed to the Levenberg-Marquardt getting stuck in a local minimum while optimising. As the network is technically a statistical model rather than an iterative optimisation, then the possibility for local minima is avoided.

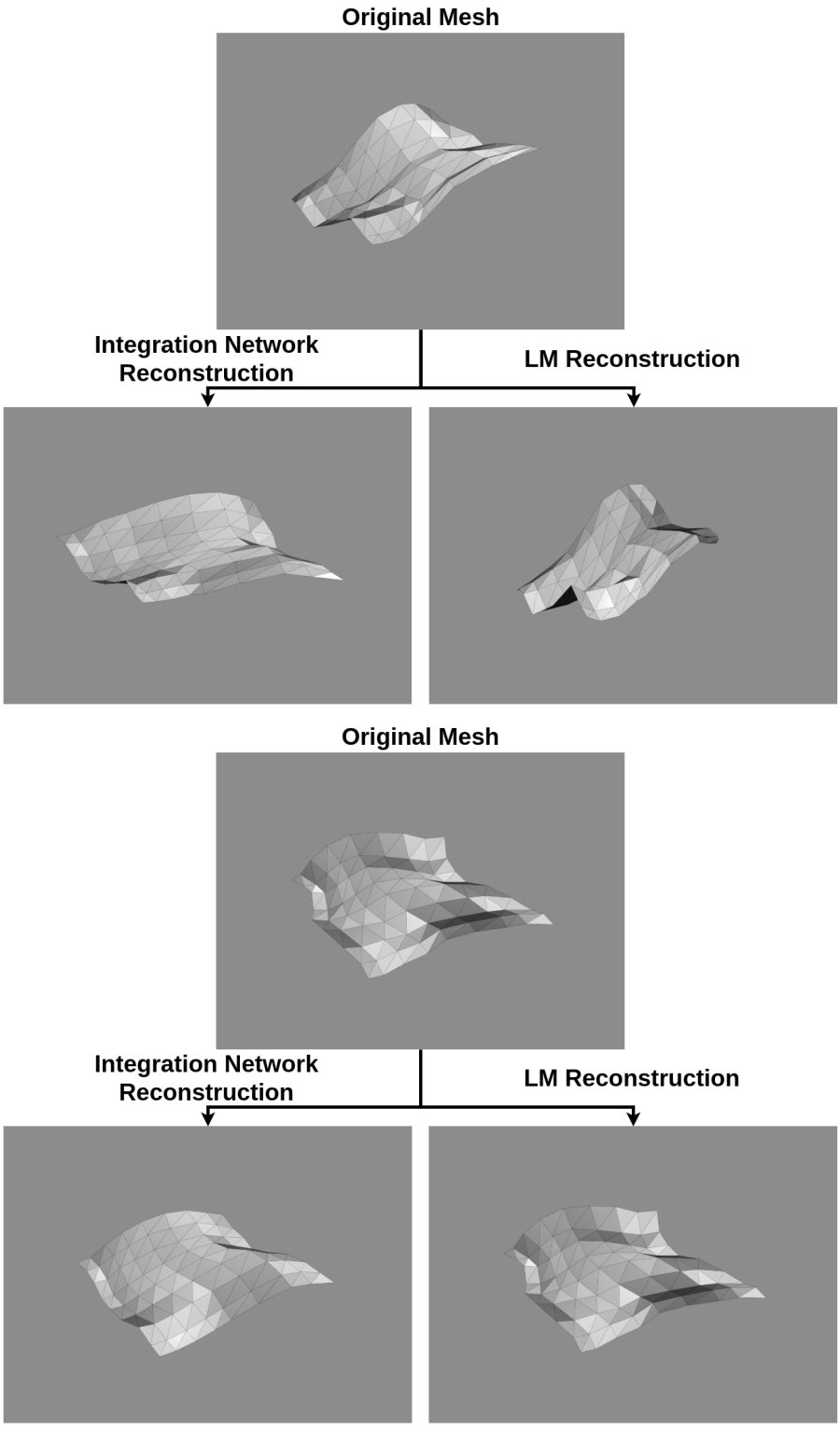


Figure 6.4: Two examples comparing integration network with Leverberg-Marquardt output for 8x8 meshes. The same observation holds for 8x8 meshes where Leverberg-Marquardt output varies much more in reconstruction quality than the integration network.

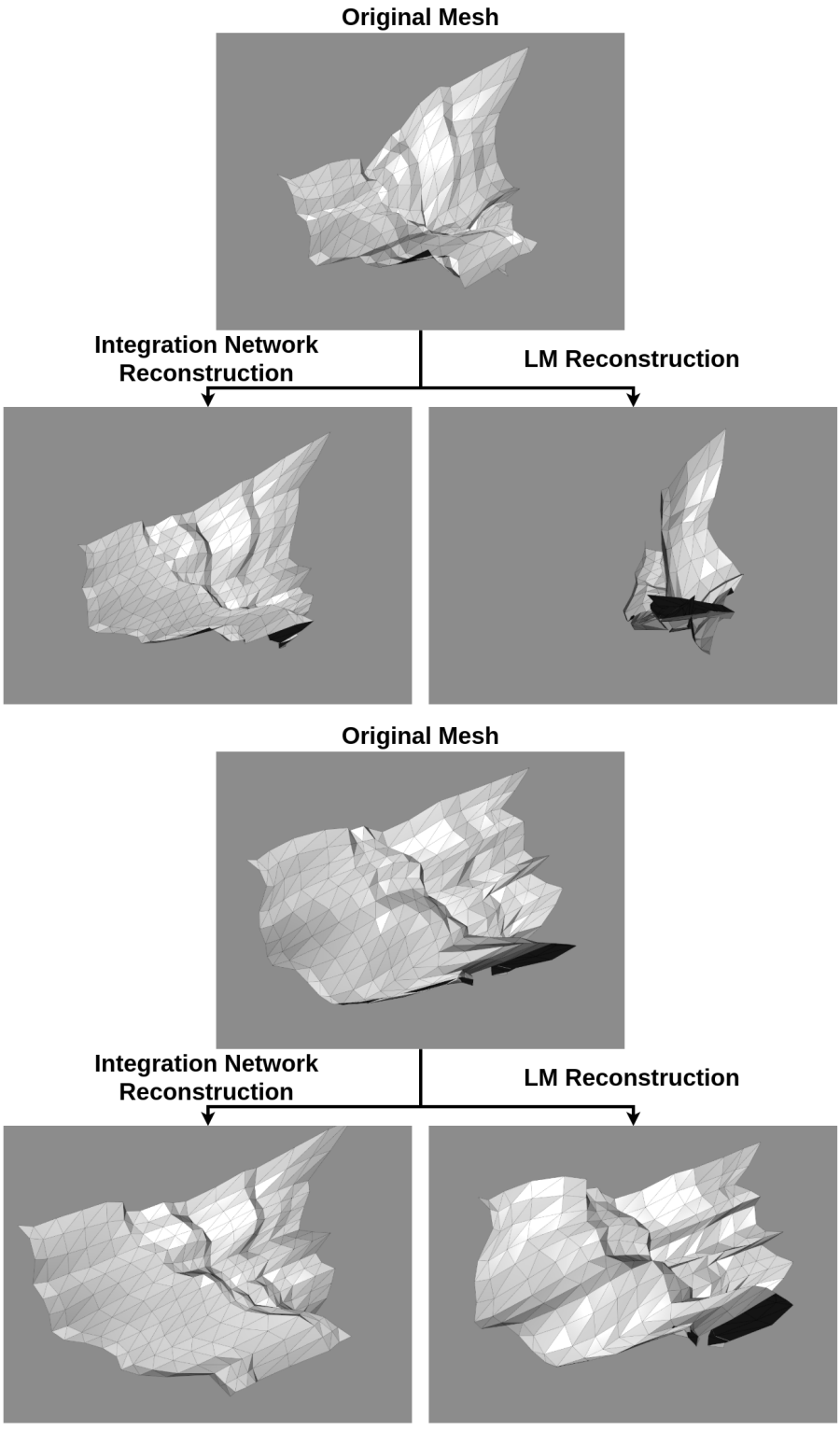


Figure 6.5: Two examples comparing integration network with Leverberg-Marquardt output for 16x16 meshes. The same observation holds for 16x16 meshes where Leverberg-Marquardt output varies much more in reconstruction quality than the integration network.

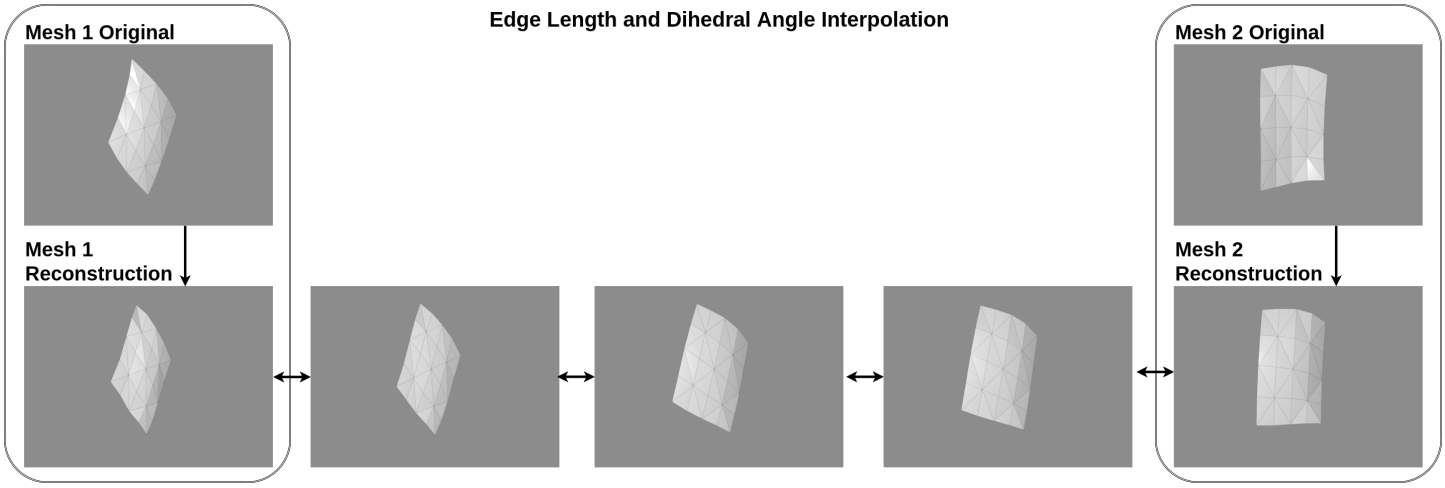


Figure 6.6: This figure shows an interpolation between two 4x4 meshes in edge length and dihedral angle form. Two original meshes were converted to this form, and then a linear interpolation is done between them. At each step, the result is reconstructed using the integration network and displayed. Interpolations are smooth and intuitive, showing that the integration network allows us to perform mesh interpolation in edge and dihedral angle space without having to constantly re-optimise as would be required if interpolating numerically.

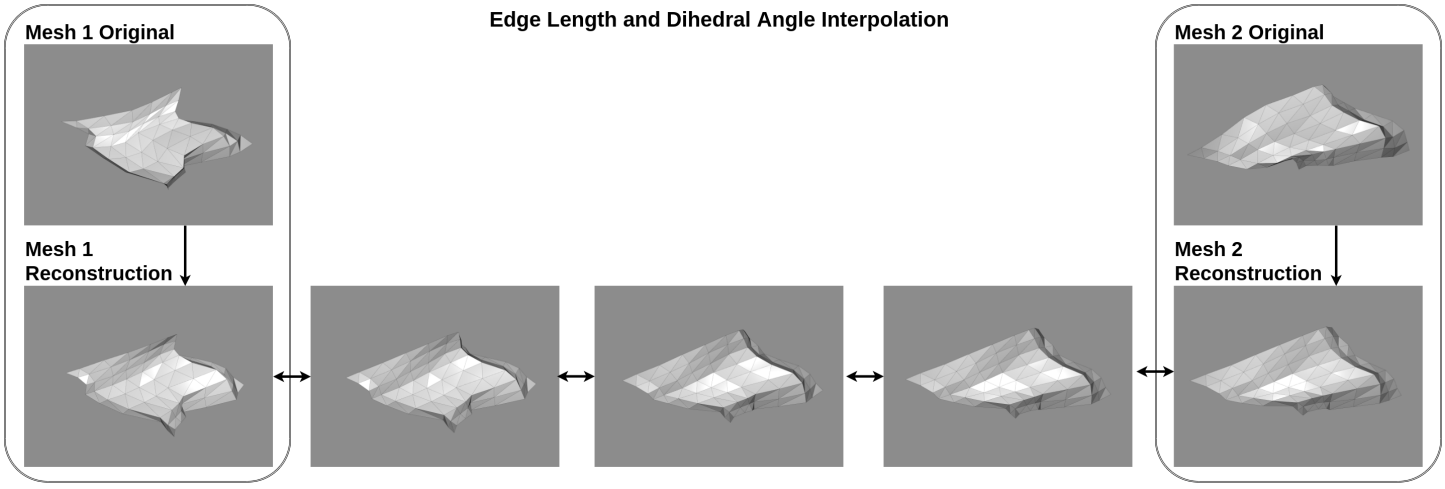


Figure 6.7: This figure shows an interpolation between two 8x8 meshes in edge length and dihedral angle form. Interpolation is done in the same way as for 4x4 meshes. The interpolation quality seen for 4x4 is preserved with 8x8 meshes.

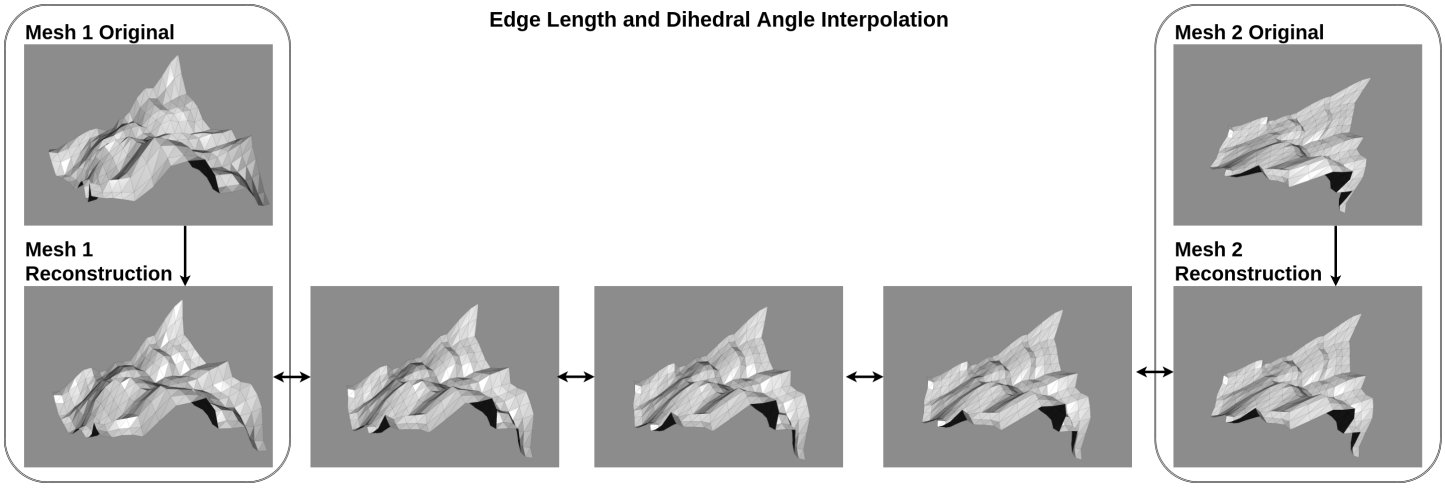
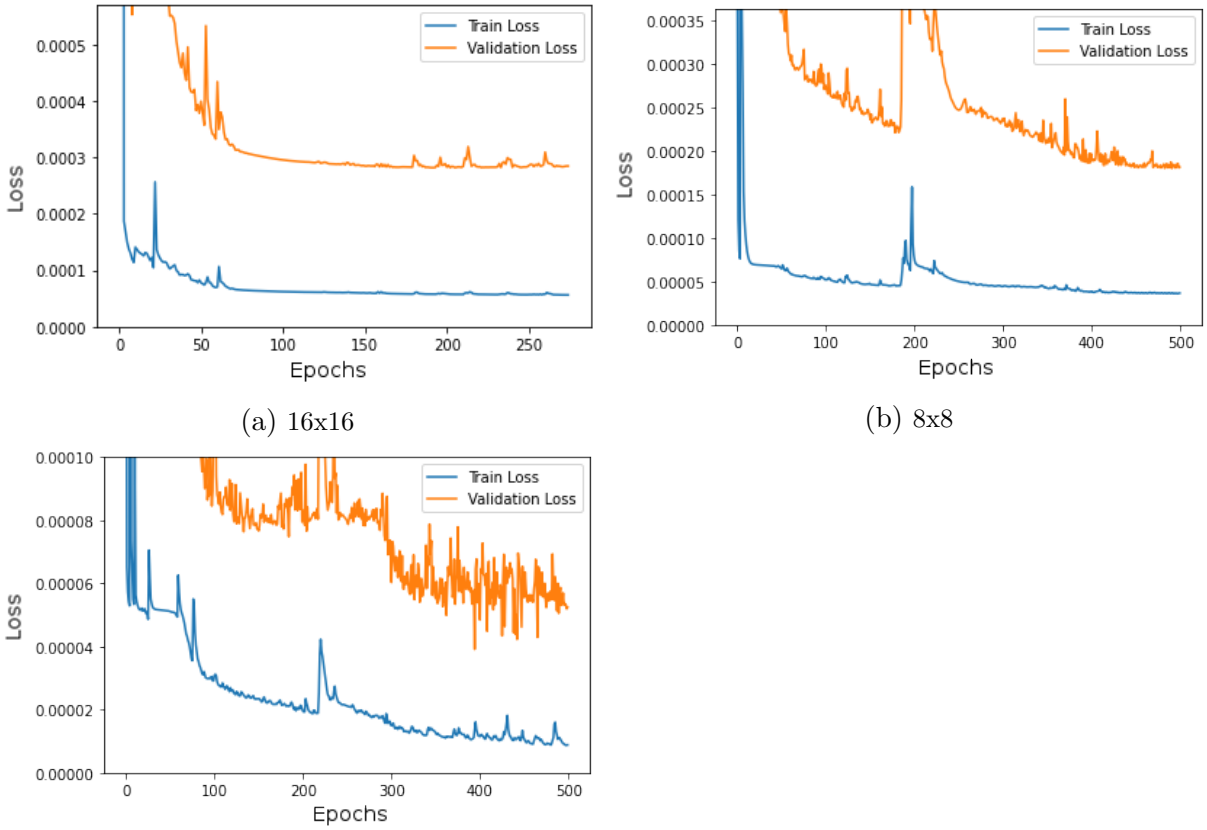


Figure 6.8: This figure shows an interpolation between two 16x16 meshes in edge length and dihedral angle form. Interpolation is done in the same way as for 4x4 meshes. The interpolation quality seen for 4x4 is preserved with 16x16 meshes.

6.2 Fixed Point/Plane Constraints

In this section we will look at the performance of the autoencoder framework, and its ability to generate new meshes which adhere to constraints fixing points to certain positions or planes. Experiments were performed as outlined in chapter 5.2.



(a) 16x16

(b) 8x8

(c) 4x4

Figure 6.9: Losses recorded while training the autoencoder on each data set. Data sets were the same as the ones used for the integration network, with the same training/test split. Loss was measured using PyTorch's mean squared error loss function, comparing autoencoder input and output. The 4x4 dataset produced the best loss value and the 16x16 the worst, although 16x16 and 8x8 were relatively similar. For the 16x16 dataset, training was only done for 250 epochs instead of 500, because after this point both the training and test losses increased dramatically.

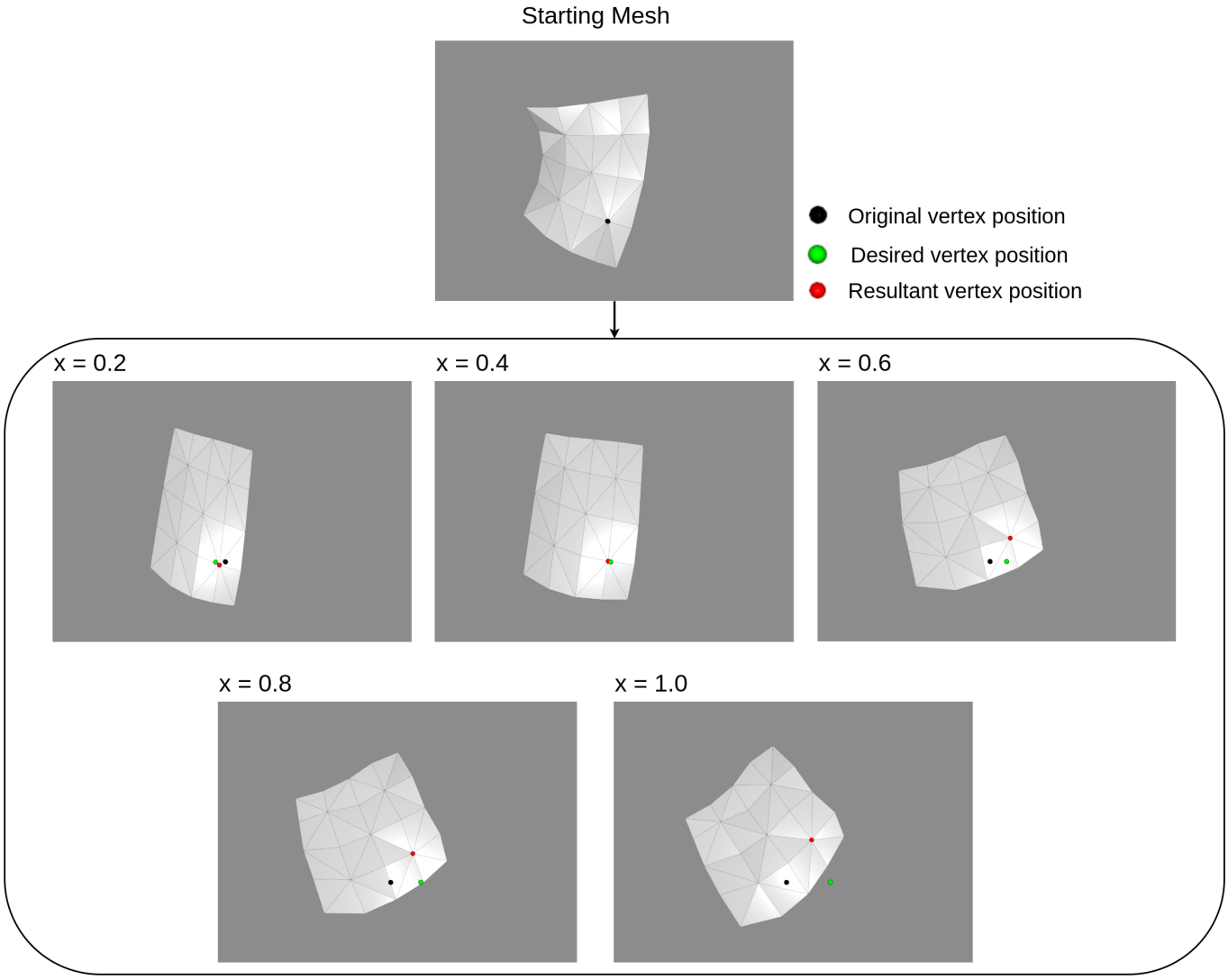


Figure 6.10: Examples for generating a 4x4 mesh with a single fixed vertex. In each example, the x coordinate of the fixed point is changed and the y, z coordinates are kept the same. The position that we are fixing the point to is shown in green, and the actual position of that vertex is shown in red. For values of x close to zero, fixing the point is achieved with the best precision.

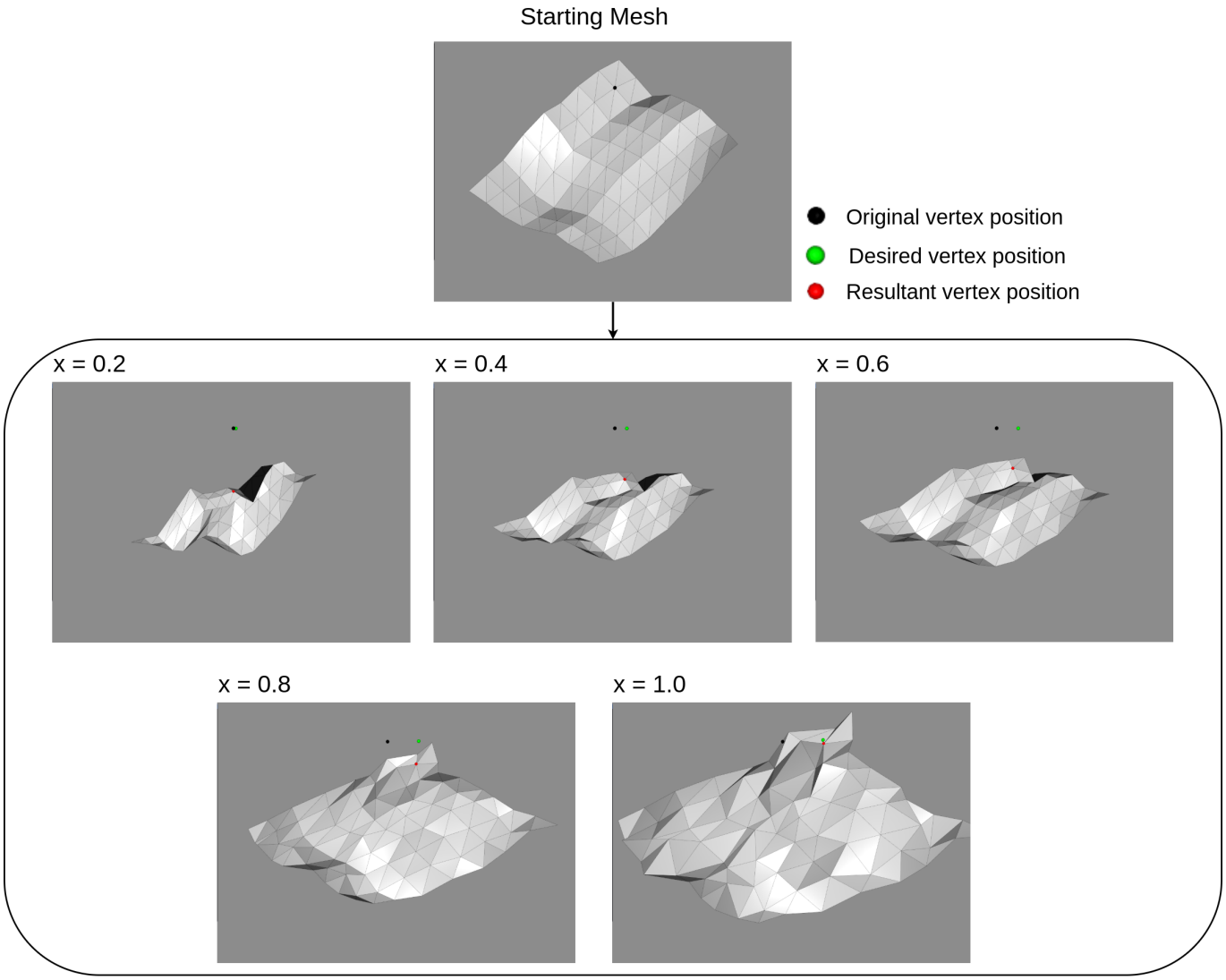


Figure 6.11: Examples for generating an 8x8 mesh with a single fixed vertex. In each example, the x coordinate of the fixed point is changed and the y, z coordinates are kept the same. The position that we are fixing the point to is shown in green, and the actual position of that vertex in the mesh is shown in red. Converse to what was seen in the 4x4 data set, fixing the point worked better for values of x further from zero.

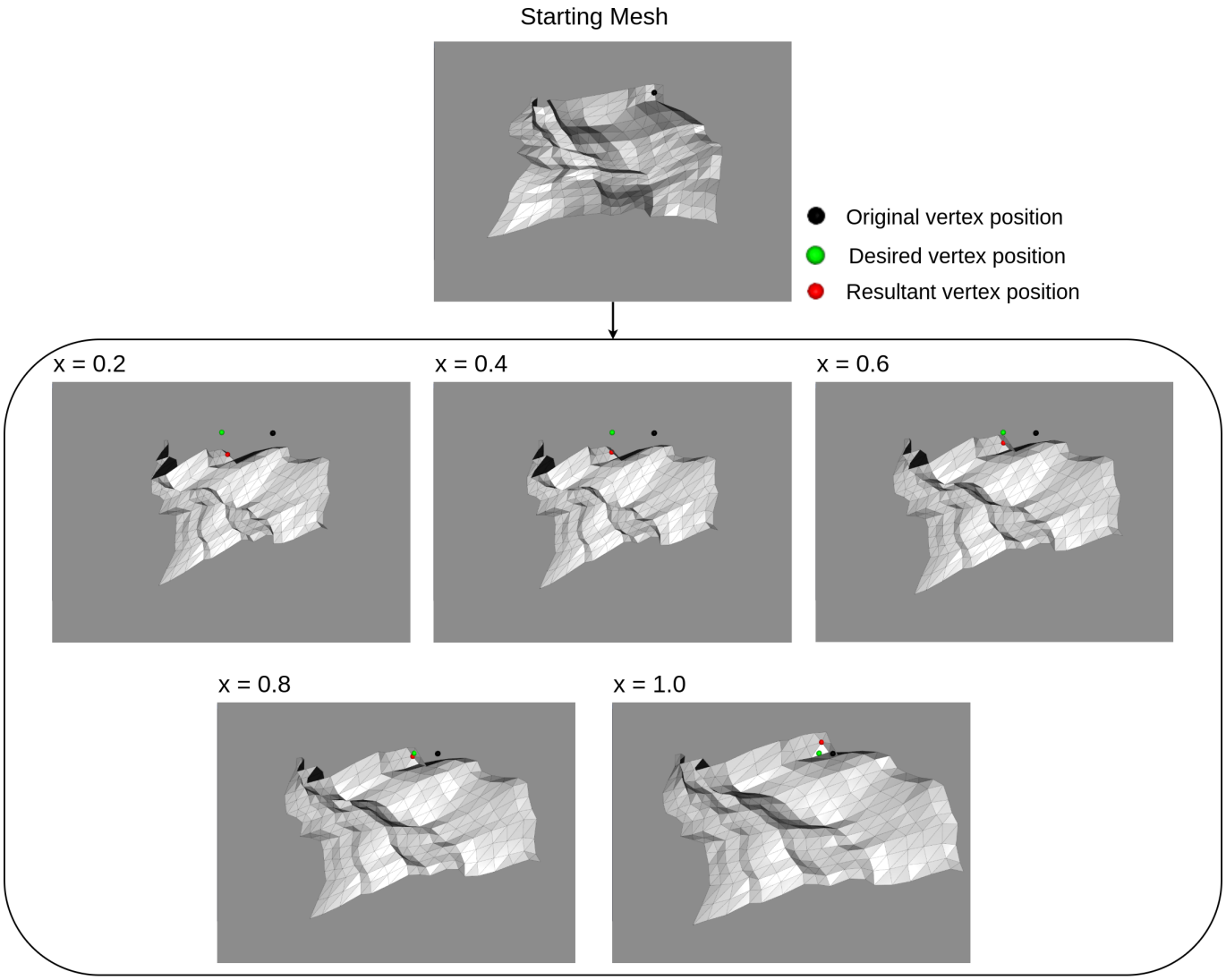


Figure 6.12: Examples for generating a 16x16 mesh with a single fixed vertex. In each example, the x coordinate of the fixed point is changed and the y, z coordinates are kept the same. The position that we are fixing the point to is shown in green, and the actual position of that vertex in the mesh is shown in red. The results for the 16x16 mesh are more comparable to the 4x4 mesh. Fixing a point achieved higher precision when the x coordinate was closer to zero.

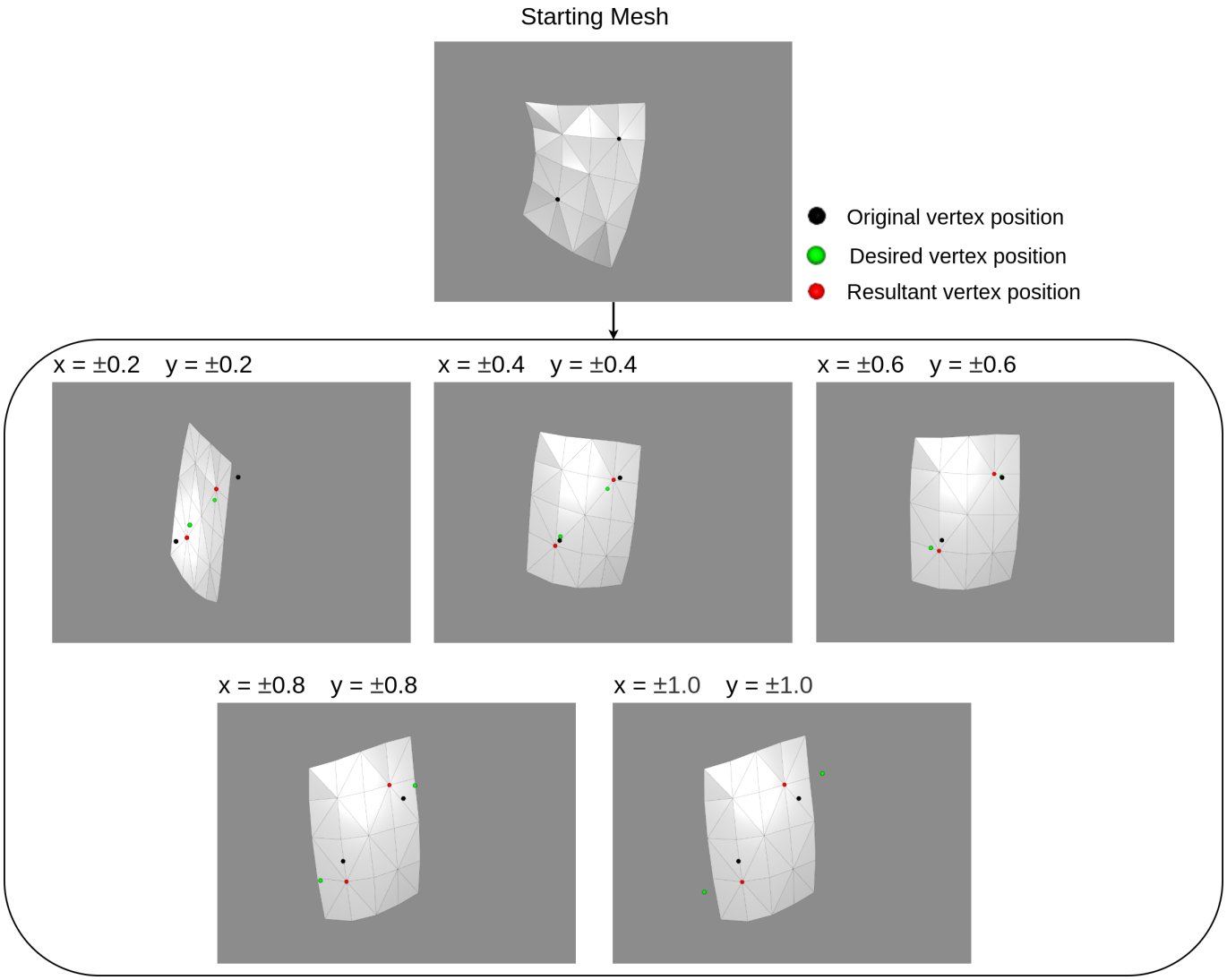


Figure 6.13: Examples for generating a 4x4 mesh with two fixed vertices. In each example, the fixed points are given different x and y values, while the z coordinates are kept constant. Fixing two points gives relatively good results for x and y values close to zero.

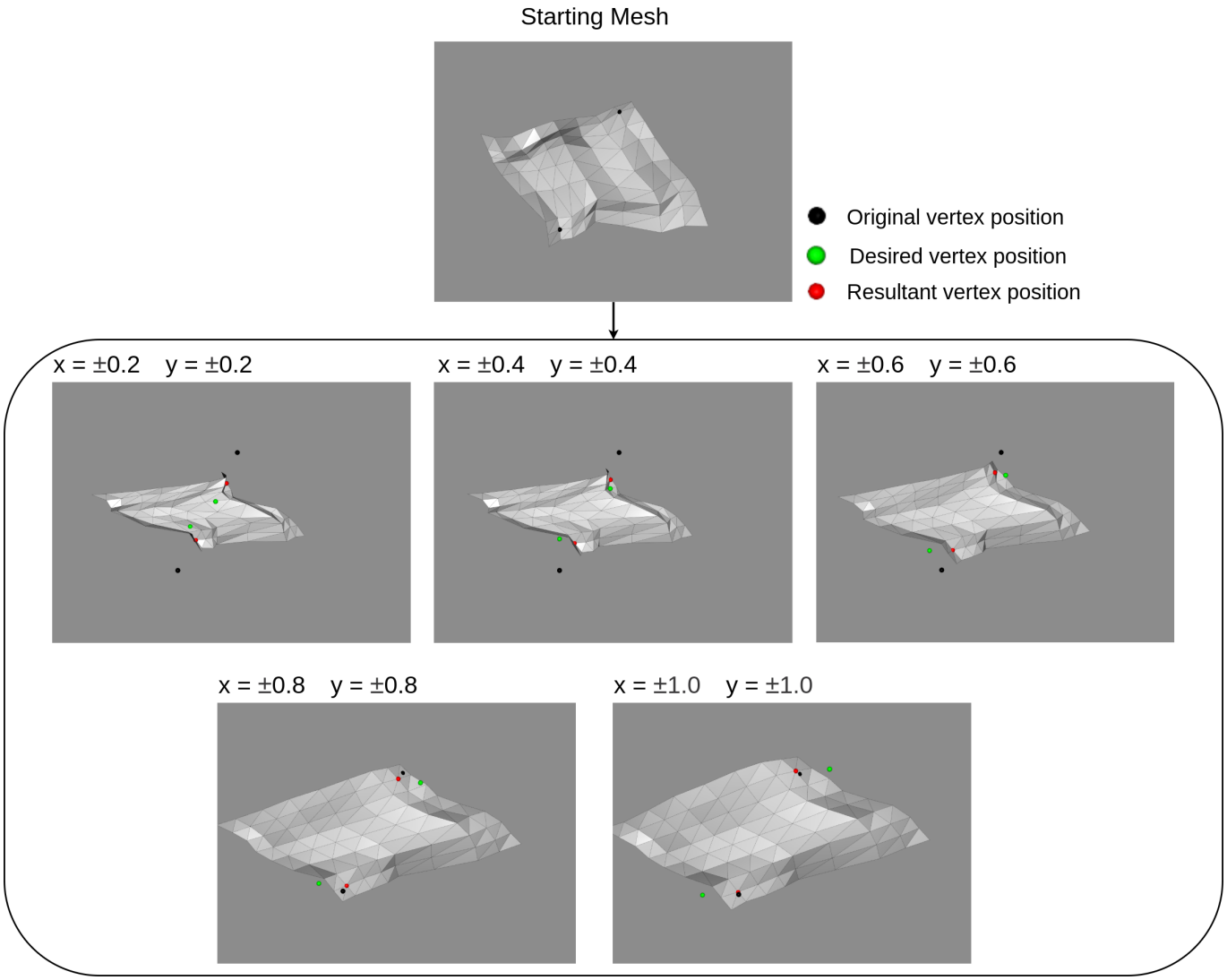


Figure 6.14: Examples for generating an 8x8 mesh with two fixed vertices. In each example, the fixed points are given different x and y values,while the z coordinates are kept constant. As with the 4x4 examples, the fixed points are most accurate at values of y and x close to zero.

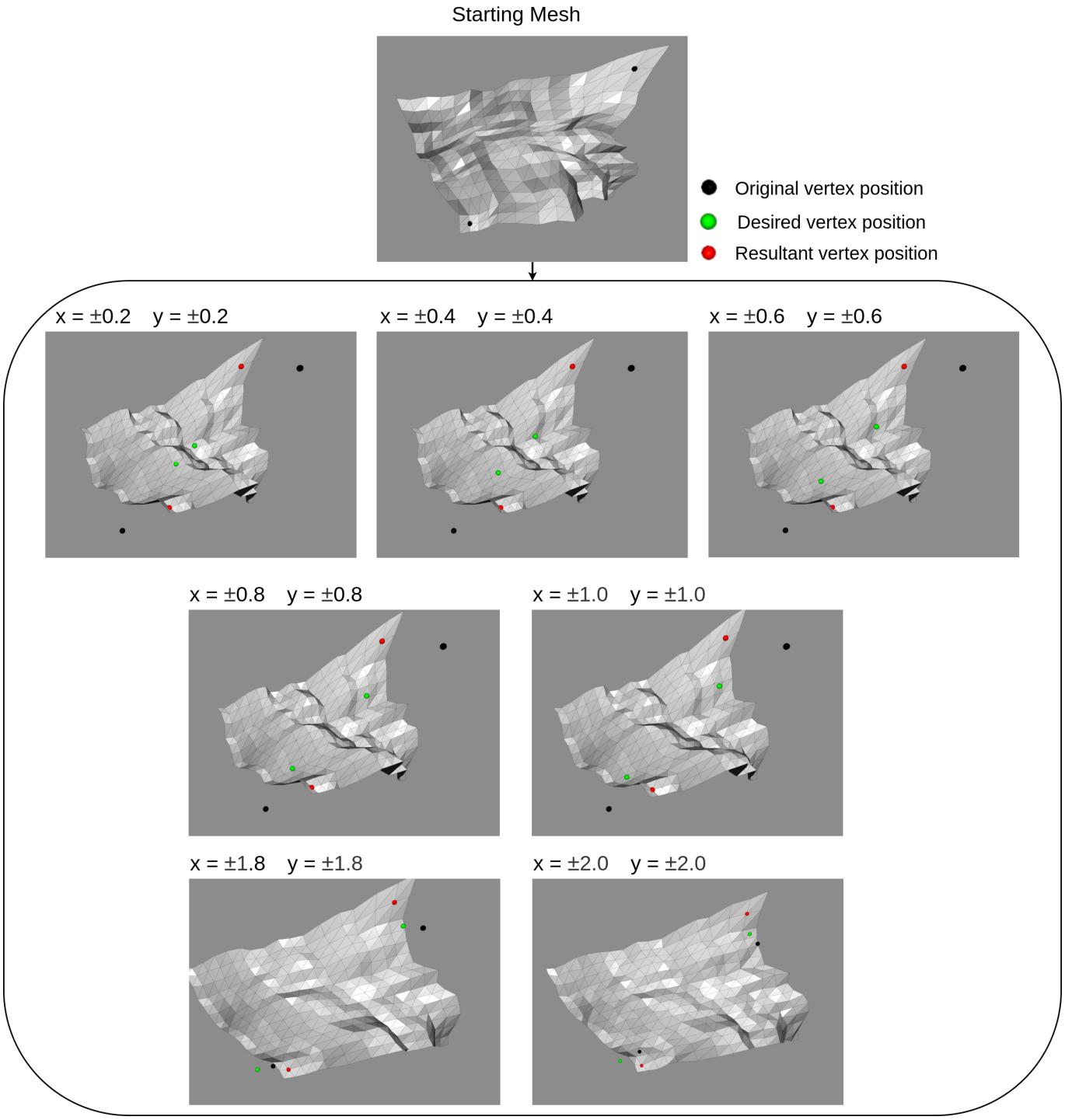


Figure 6.15: Examples for generating a 16x16 mesh with two fixed vertices. In each example, the fixed points are given different x and y values, while the z coordinates are kept constant. From the examples it is clear that the latent space did not capture meshes with x and y values close to zero. As such, some extra examples are included to show that fixing the points becomes more possible as the fixed positions approach the starting vertex positions.

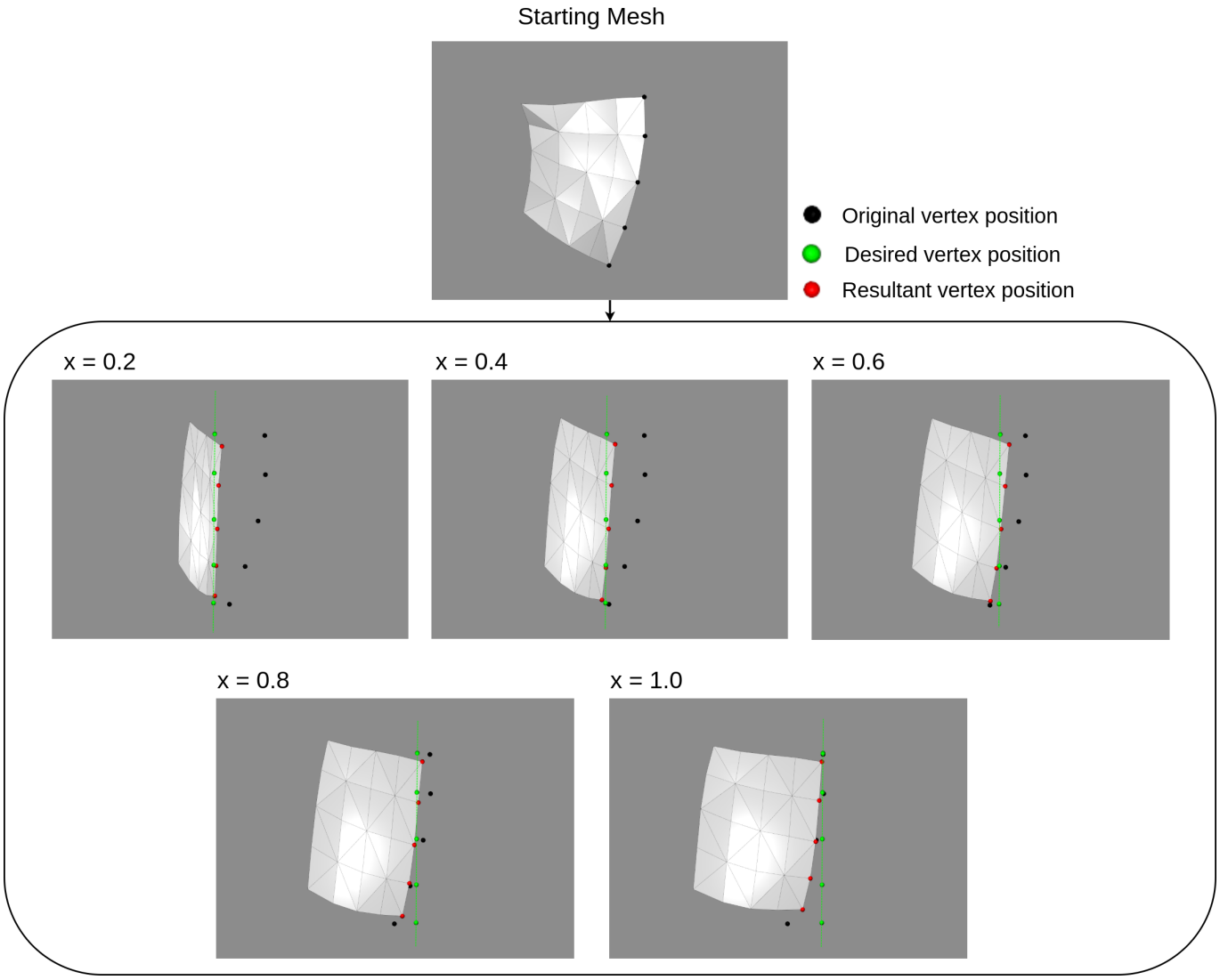


Figure 6.16: Examples of fixing a row of vertices to a single plane on a 4x4 mesh. Y and z coordinates for each vertex in the row were not constrained, only x coordinates were constrained to a specific value. The goal is for all the vertices denoted in red to be as close as possible to the green line. For values of x closer to zero, the margin of error on the fixed plane was smaller

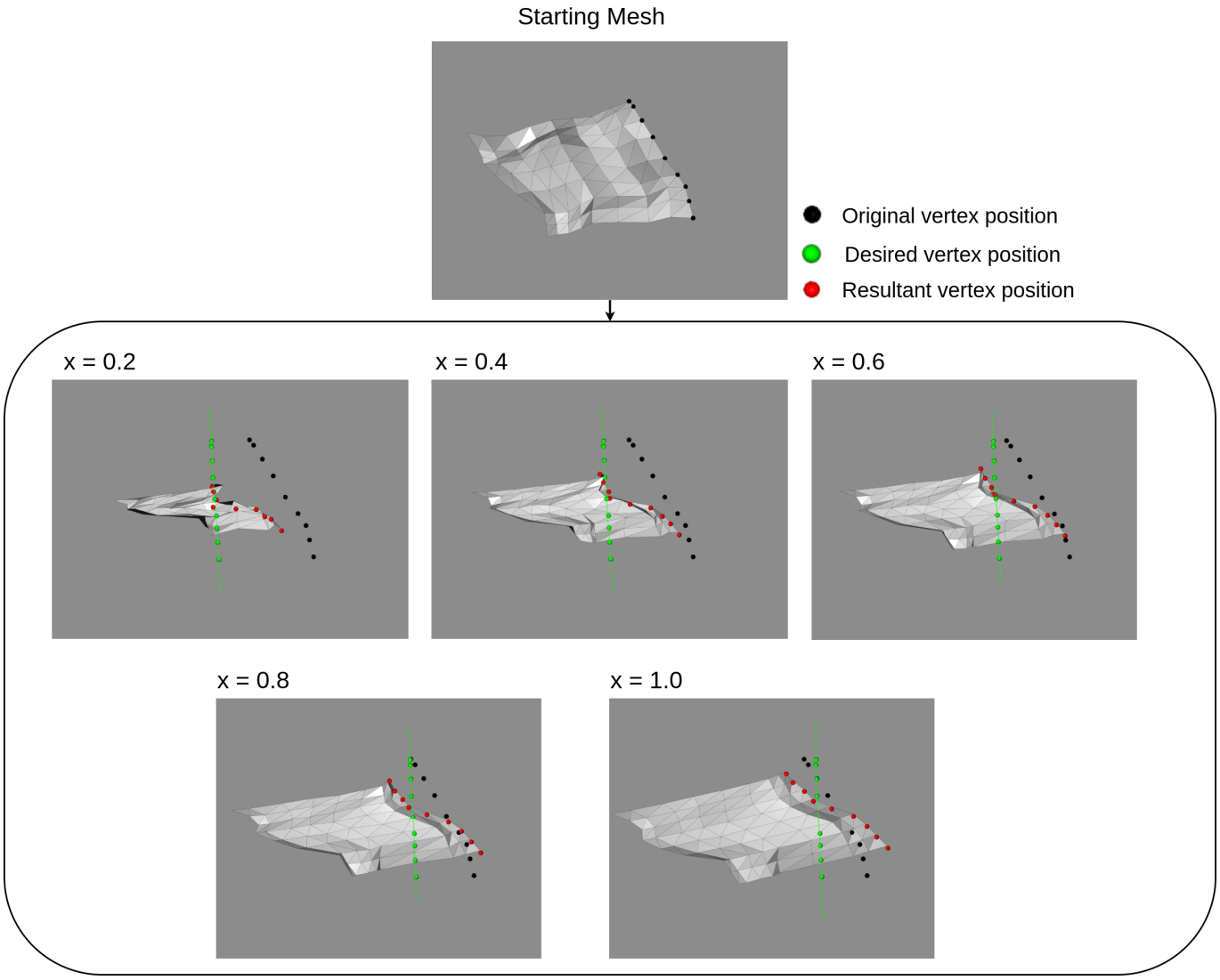


Figure 6.17: Examples of fixing a row of vertices to a single plane on a 8x8 mesh. Y and z coordinates for each vertex in the row were not constrained, only x coordinates were constrained to a fixed value. The goal is for all the vertices denoted in red to be as close as possible to the green line. As can be seen, the fixed points were not able to be properly lined up on the required plane. At best, some examples showed a subset of the row successfully lined up on the plane

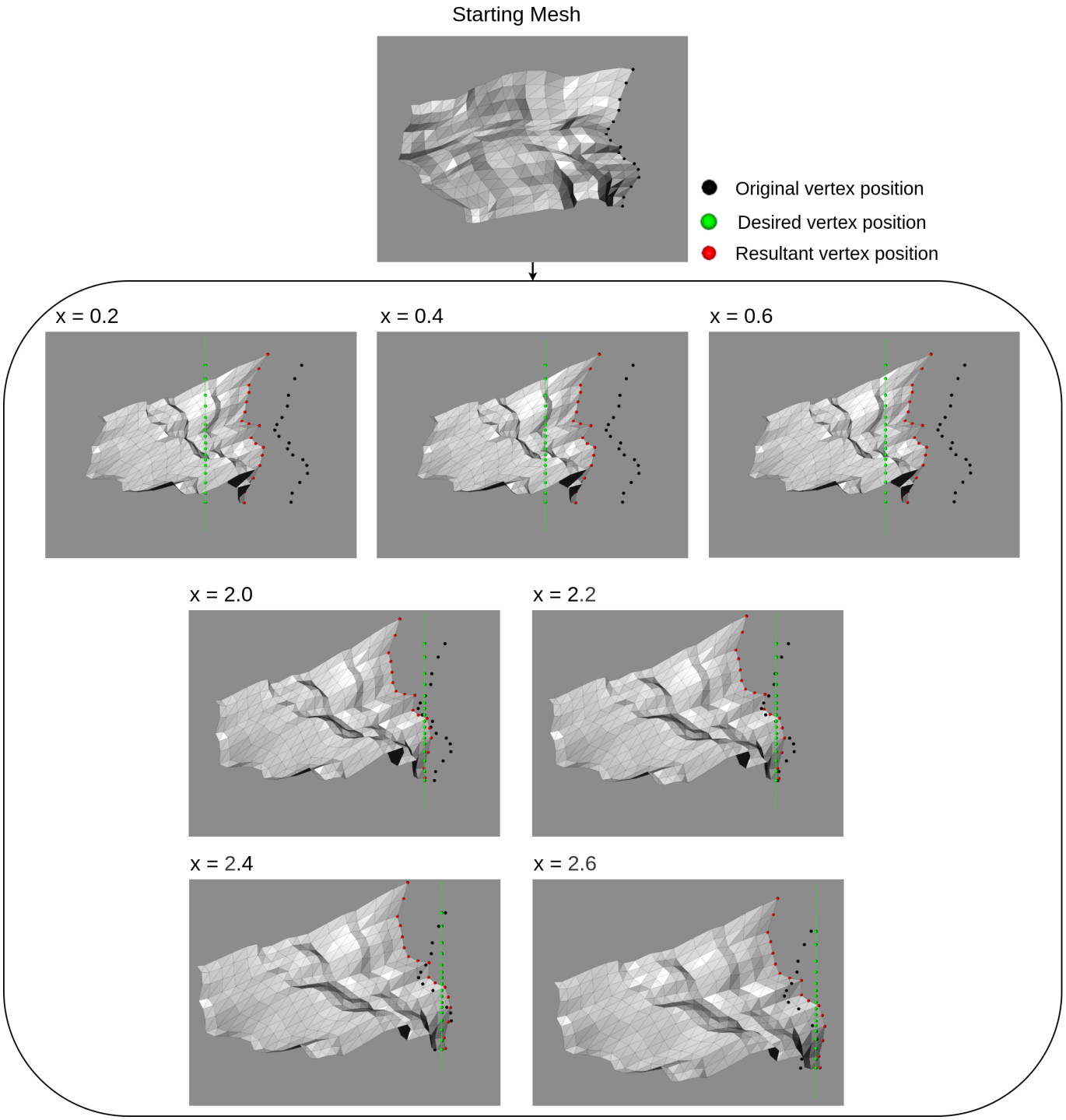


Figure 6.18: Examples of fixing a row of vertices to a single plane on a 16x16 mesh. Y and z coordinates for each vertex in the row were not constrained, only x coordinates were constrained to a specific value. The goal is for all the vertices denoted in red to be as close as possible to the green line.. Results are relatively comparable to the 8x8 example- the entire row was unable to be fixed to a single plane. For each example though, a subset of the row does lie close to the plane.

Chapter 7

Discussion

The original aim of this project was to develop a generative neural network which could generate constrained three dimensional quad meshes. The scope of the project covered constraints which fixed a designated vertex or set of vertices to a specific plane or points. An important consideration however, was to develop a framework that is as general as possible to allow for extension into other types of constraints in the future. After some unsuccessful initial experimentation using vertex coordinates, we chose to use a mesh's set of edge lengths and dihedral angles as input for our framework. This representation was used by Fröhlich and Botsch[FB11] and chosen for our framework as it captures the geometry of quad mesh better than other, more naive representations.

Using the edge lengths and dihedral angles is useful in capturing the geometry of a mesh for a neural network, but is not useful when rendering a mesh. This led to the secondary goal of the project- using a neural network to convert between our edge length and dihedral angle format, to a list of vertex positions of a mesh. This integration problem would typically be solved using an iterative optimisation algorithm, however our idea was that using a network would be much faster and could then be built into our generative framework.

7.1 Integration Network

The first part of the experiments was focused on the integration network. The performance of the integration network was measured in comparison with an implementation of the Levenberg-Marquart algorithm in MATLAB. Results showed that for all three mesh sizes tested the integration network outperformed Levenberg-Marquardt, producing reconstructions with better accuracy In observing the renderings of specific examples used for experiments however, it would be too simplistic to say that the integration network works better than Levernberg-Marquardt. Reconstruction examples done using Levenberg-Marquardt tended to either be extremely accurate, or extremely inaccurate. Conversely, the integration network did not produce reconstructions to the same accuracy, but did so consistently and did not suffer from the extremely inaccurate outliers.

The difference in the way that the two integration methods perform leads to a trade off between consistency, and accuracy between the methods. The integration network provides a way to convert from edge lengths and dihedral angles, to vertex coordinates very quickly, reliably and with a reasonable accuracy. This makes it ideal for use with applications that require a large number of reconstructions, very fast reconstruction or do not require extreme accuracy. On the other side of the trade off, Levenberg-Marquart produces reconstructions with extremely high accuracy most of the time, but takes longer for each reconstruction and has a chance to produce an extremely poor result. Applications which only require a small number of reconstructions, or for which accuracy is the most important factor are more suited to using Levenberg-Marquart.

The intended application for the our constrained mesh generation framework is for

incorporation in an architecture design tool or something similar. For such design tools, speed is important so that a user can interact with a mesh in real time and manipulate it without having to wait for the application to process their inputs. Such a requirement means that the integration network is the better suited reconstruction method for our framework.

As an extra application of the integration network, we performed experiments to show that the integration network allows for smooth linear interpolations between meshes on the edge lengths and dihedral angles. While this is technically possible using an optimisation such as Levenberg-Marquardt, it requires re-optimising at every intermediate step and runs the risk of hitting a poor outlier as we have seen in the comparison experiments. The consistency in its accuracy and the fact that we do not have to perform an entire optimisation at every step, allows for easy and smooth interpolation in the edge length and dihedral angle space- an application which is useful in architecture design.

7.2 Constraints

The second set of experiments tested the generation of constrained meshes using the autoencoder framework. Three variations on the fixed point constraints were tested- fixing a single vertex to a point, fixing two vertices to two separate points and fixing a row of vertices to a single plane. Moderate results were achieved overall but accuracy and trends did vary amongst the three constraint variations as well as amongst the different mesh sizes that were experimented with.

7.2.1 Constraints - Single Fixed Point

For each of the three mesh sizes used for experiments, there was a distinct region for which the single point could be fixed to. This region was not consistent between the three sizes however. For generating 4x4 meshes, it was possible to fix the point to a small region surrounding the position of the vertex in the starting mesh (initial guess). For 8x8 meshes the region where the vertex could be successfully fixed was for more extreme values for the x coordinate around 1.0, and for 16x16 the region was relatively similar to the 4x4 meshes.

It should be noted that for every example- even those that did not produce a fixed vertex anywhere close to the desired point, there was always at least one axis where the actual vertex position and its desired position were very close. This is an indication that the latent space optimisation was working correctly, and that the output provided a vertex position as close to the desired position as possible. The limitations instead likely stem from the actual range of meshes spanned by the latent space. The range of possible meshes that can be generated from the latent space reflects the range of meshes in the training data. As such, if a specific vertex position is not sufficiently represented in the training data then constraining the vertex to that position will not be successful.

This limitation stemming from the span of the latent space also could explain why the behaviour of the 8x8 mesh was different from the 4x4 and 16x16 meshes. If the example chosen as the starting mesh contained vertex positions that were very different from the positions represented in the training data, then the latent space could not produce meshes with such vertex positions. Since we choose fixed positions relative to the starting mesh, this could explain the situation that we see where points close to the original vertex cannot be fixed, but more extreme positions can.

7.2.2 Constraints - Two Fixed Points

Similar to fixing a single vertex, when fixing two vertices to separate points there was a distinct region where this worked best. For the 4x4 and 8x8 meshes, these regions seemed much larger than when fixing a single point, but this was not the case for 16x16 meshes. The way that the fixed vertex positions were chosen for each experiment was like squeezing and stretching the starting mesh along a diagonal line. For the cases of the 4x4 and the 8x8 meshes, the region where the points could be fixed best showed the limits of how far the mesh could be squeezed or stretched. This is likely to be the case for the 16x16 mesh as well, but when using the same fixed points as the other experiments almost the identical output was produced every time. Following the observations from the other sizes, it's possible that these points were already squeezing the 16x16 much as much as possible. Following this logic, two further examples are shown for the 16x16 meshes showing fixed points with more extreme values. For these examples, the constraints were enforced with more success.

The results for two fixed points are in line with the idea that the span of the latent space is the limiting factor to successfully generating constrained meshes with fixed points. If the training data contained a larger range of positions for the two constrained vertices, then likely the generated meshes could be squeezed and stretched further in order to better satisfy constraints.

7.2.3 Constraints - Fixed Plane

The third constraint variant that was tested was constraining a row of vertices to a single plane. We saw that this was reasonably achievable for smaller meshes of size 4x4, however for 8x8 and 16x16 meshes at best only a subset of the row could be constrained to the plane. The latent space limitation as explained for the previous two constraints can also be applied here to explain the results. For the small 4x4 examples, the row contains comparatively few vertices and therefore it is more likely

that the latent space spans a mesh that lines this row up on a single plane. For the larger meshes, the effect of the eigenvector deformation is much more apparent and so the training data does not contain examples in which an entire row lines up to a single plane. This means that the latent space will not span such a mesh and so the best that the latent space optimisation can do is find a mesh that has a subset of the row lining up to the plane.

7.3 Summary

Overall, some moderate success was seen in generating meshes for each of the three fixed point constraint variants. This success however, is very limited in the range of positions that we are able to constrain a set of points to, and this range is not necessarily similar between different mesh sizes. The eigenvector deformation method that we used to create our training data, was intended to produce a general set of deformed meshes that could be easily encoded into a latent space. This was successful, we were able to create an integration network that accurately converts edge length and dihedral angles to vertex coordinates and then encodes a span of meshes in a latent space. Despite this, using these meshes as training data also served as a limitation in generating meshes constrained with fixed vertices. Creating training data by deforming meshes using our eigenvalue method meant that in the training data sets, the range of positions that a single vertex took across the different meshes was quite small. This range was encoded into the latent space, ultimately limiting the region to which vertices could successfully be constrained.

If we could use a different and more varied data set and still achieve similar loss in training the integration network and autoencoder, it's very likely that results for the fixed point constraints would be improved. The problem is that using such a

data set may be counter-intuitive to our goal of keeping the autoencoder general and extensible to other types of constraints. The eigenvector method of deformation was chosen specifically with extensibility in mind and so moving away from the eigenvectors could compromise that. Another approach to improve the constrained mesh generation could be to find a neural network architecture for the integration network and autoencoder, that could somehow omit any bias towards the eigenvector deformations. If the networks could solely learn based on the geometric information of the data and resist any bias regarding the eigenvectors, then the latent space would likely be far more robust in generating constrained meshes.

Chapter 8

Conclusion

In this thesis, we created an autoencoder neural network architecture trained to generate constrained quad meshes. Our network is capable of generating meshes with fixed point constraints, fixing a single or set of vertices to a specific set of points or plane. The goal was to create a framework that could feasibly be integrated into some sort of architectural design tool. Through the process of creating this autoencoder, we have also achieved a secondary goal of training a multi-layer perceptron neural network to convert a quad mesh represented by a set of edge lengths and dihedral angles into a set of vertex coordinates. This conversion is an integration which is typically solved using a non-linear least squares optimisation algorithm.

The integration network we created for integrating between mesh representations presents a new way to solve such an integration, with a number of benefits over numerical solutions. Results showed that overall, the integration network completed the conversion with lower error than an implementation of the non-linear least squares Levenberg-Marquardt algorithm, however a trade off exists between the two methods. The integration network produces more consistent results however Levenberg-Marquardt is capable of higher accuracy. Despite being capable of higher accuracy though, Levenberg-Marquardt also often produced results of extremely poor quality. Different applications will benefit from the different advantages of the two methods

but since our application values speed and consistency highest, the integration network is more fit for our use.

Generating constrained meshes showed moderate results. In some cases meshes could be generated which adhered closely to the fixed point constraints but in other cases this was not possible at all. For each data set a rough region where points could be constrained was identified, and this reflected the range of data that the autoencoder was trained on. Further improvements could be made by either finding an architecture that can successfully train on a broader data set, or an architecture that more accurately learns the mesh geometry without bias towards the eigenvectors used to create the data set.

Bibliography

[AW11] Marc Alexa and Max Wardetzky. "Discrete Laplacians on general polygonal meshes". In: *ACM SIGGRAPH 2011 papers*. ACM SIGGRAPH 2011, 2011, pp. 1–10.

[Bok+12] Martin Bokeloh et al. "An algebraic model for parameterized shape editing". In: *ACM Transactions on Graphics (TOG)* 31.4 (2012), p. 78.

[Cra18] Keenan Crane. "Discrete differential geometry: An applied introduction". In: *Notices of the AMS, Communication* (2018), pp. 1153–1159.

[Den+13] Bailin Deng et al. "Exploring local modifications for constrained meshes". In: *Computer Graphics Forum*. Vol. 32. Wiley Online Library. 2013, pp. 11–20.

[Den+15] Bailin Deng et al. "Interactive design exploration for constrained meshes". In: *Computer-Aided Design* 61 (2015), pp. 13–23.

[FB11] Stefan Fröhlich and Mario Botsch. "Example-driven deformations based on discrete shells". In: *Computer graphics forum*. Vol. 30. Wiley Online Library. 2011, pp. 2246–2257.

[Ful+19] Lawson Fulton et al. "Latent-space Dynamics for Reduced Deformable Simulation". In: *Computer Graphics Forum*. Vol. 38. Wiley Online Library. 2019, pp. 379–391.

[Hee+16] Behrend Heeren et al. "Splines in the space of shells". In: *Computer Graphics Forum*. Vol. 35. Wiley Online Library. 2016, pp. 111–120.

[Hee+18] Behrend Heeren et al. "Principal Geodesic Analysis in the Space of Discrete Shells". In: *Computer Graphics Forum*. Vol. 37. Wiley Online Library. 2018, pp. 173–184.

[HK12] Martin Habbecke and Leif Kobbelt. "Linear analysis of nonlinear constraints for interactive geometric modeling". In: *Computer Graphics Forum*. Vol. 31. Wiley Online Library. 2012, pp. 641–650.

[Li+17] Jun Li et al. "Grass: Generative recursive autoencoders for shape structures". In: *ACM Transactions on Graphics (TOG)* 36.4 (2017), p. 52.

[Luo+18] Ran Luo et al. "Deepwarp: Dnn-based nonlinear deformation". In: *arXiv preprint arXiv:1803.09109* (2018).

[Mas+15] Jonathan Masci et al. “Geodesic convolutional neural networks on riemannian manifolds”. In: *Proceedings of the IEEE international conference on computer vision workshops*. 2015, pp. 37–45.

[Mo+19] Kaichun Mo et al. *StructureNet: Hierarchical Graph Networks for 3D Shape Generation*. 2019. arXiv: 1908.00575 [cs.GR].

[PCG15] Roi Poranne, Renjie Chen, and Craig Gotsman. “On linear spaces of polyhedral meshes”. In: *IEEE transactions on visualization and computer graphics* 21.5 (2015), pp. 652–662.

[Pot+15] Helmut Pottmann et al. “Architectural geometry”. In: *Computers & graphics* 47 (2015), pp. 145–164.

[Qi+17] Charles R Qi et al. “Pointnet: Deep learning on point sets for 3d classification and segmentation”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 652–660.

[Rad+16] Philipp von Radziewsky et al. “Optimized subspaces for deformation-based modeling and shape interpolation”. In: *Computers & Graphics* 58 (2016), pp. 128–138.

[Roc19] Joseph Rocca. *Understanding Variational Autoencoders (VAEs)*. Sept. 2019.

[Sch+14] Christian Schulz et al. “Animating deformable objects using sparse spacetime constraints”. In: *ACM Transactions on Graphics (TOG)* 33.4 (2014), p. 109.

[Sch+17] Adriana Schulz et al. “Interactive design space exploration and optimization for cad models”. In: *ACM Transactions on Graphics (TOG)* 36.4 (2017), p. 157.

[Tan+15] Chengcheng Tang et al. “Form-finding with polyhedral meshes made simple”. In: *ACM SIGGRAPH 2015 Posters*. ACM. 2015, p. 5.

[Vax12] Vaxman. “Modeling polyhedral meshes with affine maps”. In: *Computer Graphics Forum*. Vol. 31. Wiley Online Library. 2012, pp. 1647–1656.

[Vax14] Amir Vaxman. “A projective framework for polyhedral mesh modelling”. In: *Computer Graphics Forum*. Vol. 33. Wiley Online Library. 2014, pp. 121–131.

[Yan+11] Yong-Liang Yang et al. “Shape space exploration of constrained meshes”. In: *ACM Trans. Graph.* 30.6 (2011), p. 124.

[YPM12] Xin Zhao Cheng-Cheng Tang Yong, Liang Yang Helmut Pottmann, and Niloy J Mitra. *Intuitive design exploration of constrained meshes*. Citeseer, 2012.