Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione

École Polytechnique Fédérale de Lausanne
Chair of Numerical Modelling and Simulation

Innovative integration techniques on curvilinear elements

Master of Science in
Mathematical Engineering

Candidate:
Sara Zaninelli
ID Number 875728

Supervisors:
Prof. Annalisa Buffa
Prof. Paola F. Antonietti
Dr. Pablo Antolin
Ondine Chanon

Academic Year 2018 - 2019

A mio padre,
per avermi insegnato
a non mollare mai.

Abstract

The aim of this work is to find innovative techniques to approximate solutions of partial differential equations (PDEs) on trimmed domains. As a matter of fact, cutting a regular surface with a trimming curve can be an extremely powerful method to represent arbitrary surfaces, but it arises many problems, for instance the computation of integrals. Indeed, it requires a re-parametrization of the cut through Bézier functions, which involves re-meshing that leads to complex geometrical manipulations, in particular in three-dimensions (3D). Thus we propose the use of the new techniques based on machine learning (ML), especially of convolutional neural networks (CNNs), to compute mass and stiffness matrix entries and to build new suitable quadrature formulae. We want to simplify the preprocessing phase, avoiding the reconstruction of the curve. This could be done by passing as input to the neural network (NN) images showing the visible boundary of the domain. These inputs can be created easily both in two-dimensions (2D) and in 3D.

Contents

List of Figures

List of Tables

Introduction

Since when it has first been introduced in 2005, isogeometric analysis (IGA) [1] has been a successful method to convert computer-aided design (CAD) geometries to finite element (FE) computational domains. Splines have been proved to be powerful tools to approximate solutions of partial differential equations (PDEs) and a solid mathematical theory [2] has been developed over the years.

Nevertheless, the issue of the construction of CAD geometries suitable to IGA has not been completely solved, in particular because CAD represents models through the description of their boundaries rather than volume discretization [3]. In this context trimming is a very powerful tool to describe arbitrary surface boundaries; indeed the surface is cut through a curve that allows to identify superfluous areas. This means that the visualization of the surface changes, but the underlying mathematical models to construct it remain unchanged .

The cutting curve, that in principle could be any function, is often reconstructed with B-splines basis functions [4]. When we use the isogeometric approach, we need to simulate physical models directly on the trimmed patches and this needs the design of suitable integration formulae on trimmed domains. At present, the design of integration quadrature rules requires re-parameterizations and re-meshing, often resulting in laborious geometric manipulations, especially in three-dimensions (3D) and expensive integration formulae. Indeed, given a mesh, first we need to detect cut elements and transform them in the parametric space. Next, we need to reconstruct the curve, find suitable quadrature points on the reference element and pull them back in the parametric space by using suitable maps.

The aim of this work is to simplify the preprocessing phase, avoiding the reconstruction of the cutting curves and the re-meshing. This could be done through Machine Learning (ML) [24], which is a tool to find automatically inherent rules or dependencies from a large amount of data. This is a new fast growing field that is opening new perspectives in different sectors, including numerical analysis. It allows the creation of models that, given a certain input, are able to predict in a good way unseen data. The process of learning is done through the optimization of an appropriate loss function, that in many cases is the mean squared error. After considering several possibilities, the Neural Network (NN) algorithms we propose are trained using a large amount of images, that, through a discretization of the domain in pixels, indicate the position of the cut and the visible boundary. This way to describe the cutting curve is much more simple than the construction of a quadrature rule which ask for re-meshing of cut elements, especially if we want to extend it to 3D, where such re-meshing is expensive and a rather unstable process. Our numerical examples have been performed in a two-dimensional (2D) space, but the models have been created in such a way that it should be quite easy to extend them in higher dimensions, in particular to 3D. Convolutional Neural Networks (CNNs) have been proved to be the perfect tool when the input of the model consists of images. As we will explain in Chapter 2 Section 2.3, a CNN is able to extract the main features of a curve, i.e. its slopes, and it reaches a stable state in fewer iterations.

When we deal with machine learning, the most delicate part of the whole precedure is the creation of the dataset to train the model. In particular, we have to feed the model with different types of cutting curves that are able to take into account more or less all the possible scenarios. The generalization of this process to 3D is not straightforward and requires special attention. Moreover, the same problem can be faced in different ways. Indeed, in this work, we will show two different approaches to approximate solutions of PDEs. The first one aims to directly compute the mass and the stiffness matrix entries. Actually, this model performs quite well, but it is very demanding in terms of memory usage. Thus the matrices should be computed online and the time of evaluation of the model becomes an important factor to take into account. Besides it is not very easy to embed it in existing codes, like GeoPDEs [39], which relies upon the use of quadrature points. So, the goal of the second approach is to construct a new suitable quadrature formula for cut elements. The idea is that the model receives the images of the cutting curve and predicts the quadrature points and weights that are necessary to compute integrals. In this way, not only it is easy to combine its use with existing codes, but it is also less expensive in terms of memory. Moreover, the quadrature points can, in principle, be used to solve any kind of problem and this opens infinite opportunities for the employment of the method. Nonetheless, in this case the choice of an appropriate loss function must be made. Indeed, so far, we have not been able to impose either in a strong or in a weak way that the points must lay inside the visible area.

In order to provide a common background, first we will present a review on Bézier basis functions in Chapter 1, c.f. Section 1.1, and we will explain trimming and the problems we want to solve in Chapter 1, Section 1.2. Then we will show classical methods of integration over cut elements in Chapter 1, Section 1.3. We will summarize new machine learning techniques and the several types of networks that can be constructed in Chapter 2. In the second part of the thesis (Chapter 3 and Chapter 4) we will illustrate the models we have created and the different techniques we have adopted in order to compute integrals over trimmed domains in a fast and reliable way. Finally, we will draw some Conclusions.

Chapter 1

Isogeometric Analysis on trimmed geometries

In this chapter we recall the basic properties of B-splines (Section 1.1) and how they can be used to approximate complex geometries. Moreover we will explain the problem of integration over trimmed domains (Section 1.3) when trying to solve elliptic partial differential equations (Section 1.2).

1.1 Basis Functions and Curves

Following the article [4], let us consider a set of $N+1$, with $N \geq 0$ non-decreasing coordinates, called knots, $u_0 \leq u_1 \leq \ldots \leq u_N$, which are listed in a **knot vector** Ξ. A non empty half open interval $[u_i, u_{i+1})$ is called the ith **knot-span** and supports $p+1$ B-splines (Fig 1.1.1) where $p \geq 0$ is the degree of the polynomial. They are defined recursively by a strictly convex combination of B-splines of degree $p-1$, as follow: $\forall u \in [u_0, \ldots, u_N]$

$$B_{i,p}(u) = \frac{u - u_i}{u_{i+p} - u_i} B_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} B_{i+1,p-1}(u), \qquad i = 0, \ldots, I-1 \tag{1.1.1}$$

with

$$B_{i,0}(u) = \begin{cases} 1 & \text{if } u_i \leq u < u_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{1.1.2}$$

and where I is the total number of basis functions. In some cases the denominator involved in the computation of the recursive formula may become zero, then the quotient is defined to be zero by convention. Each $B_{i,p}$ in a non-empty knot-span $[u_s, u_{s+1})$ is a polynomial segment, which joins with the next one at knots. This means that the knots are the points in which the function is not $\mathbf{C}^\infty$ and, in general, the continuity between adjacent segments is $\mathbf{C}^{p-m}$. The control of continuity of a quadratic B-spline is shown in Fig. 1.1.2. The knot multiplicity m points out if successive knots are equal, i.e. $u_i = u_{i+1} = \ldots = u_{i+m-1}$. If the first and last node have multiplicity $m = p+1$, then the knot vector is called **open knot**. The classical pth-degree Bernstein polynomial is defined on the open knot

$$\Xi = \{u_0 = \ldots = u_p,\ u_{p+1} = \ldots = u_{2p+1}\}, \tag{1.1.3}$$

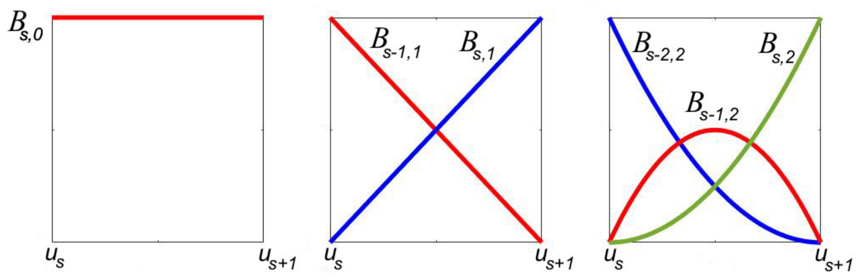


Figure 1.1.1. Non-vanishing B-splines $B_{i,p}$ on knot span s defined for $p = 0, 1, 2$.

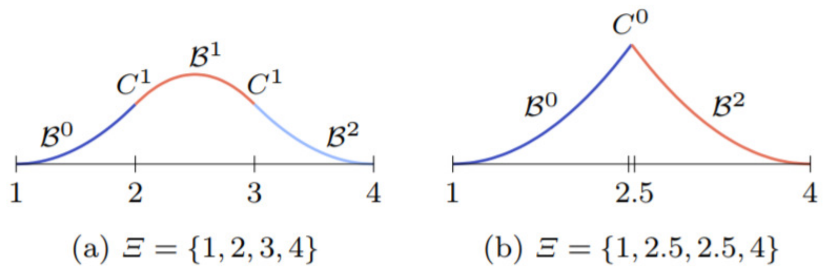


Figure 1.1.2. Polynomials segments of a quadratic B-spline, due to different knot vectors. Note the different continuity between the segments based on the knot multiplicity. Image taken from article [4].

which is usually defined in the interval $[0, 1]$, but we can avoid this restriction by a change of variables.

Given a knot vector Ξ, the B-splines based on it form a partition of unity, which means

$$\sum_{i=0}^{I-1} B_{i,p}(u) = 1, \qquad u \in [u_0, u_{I+p}] \tag{1.1.4}$$

and they are linearly independent, i.e.

$$\sum_{0}^{I-1} c_i B_{i,p}(u) = 0 \iff c_i = 0 \, \forall i \in [0, I-1]. \tag{1.1.5}$$

This means that they form a basis of the space

$$S_{p,\Xi} = \left\{ \sum_{i=0}^{I-1} c_i B_{i,p} : \; c_i \in \mathbb{R} \right\}, \tag{1.1.6}$$

so we can uniquely describe every piecewise polynomial $f_{p,\Xi}$ of degree p over a knot sequence by a linear combination of the corresponding $B_{i,p}$. A cubic B-spline basis defined by an open knot vector is visible in Fig. 1.1.3.

The first derivative of a B-spline is a linear combination of B-splines of lower degree, i.e.

$$B'_{i,p}(u) = \frac{p}{u_{i+p} - u_i} B_{i,p-1}(u) - \frac{p}{u_{i+p+1} - u_{i+1}} B_{i+1,p-1}(u). \tag{1.1.7}$$

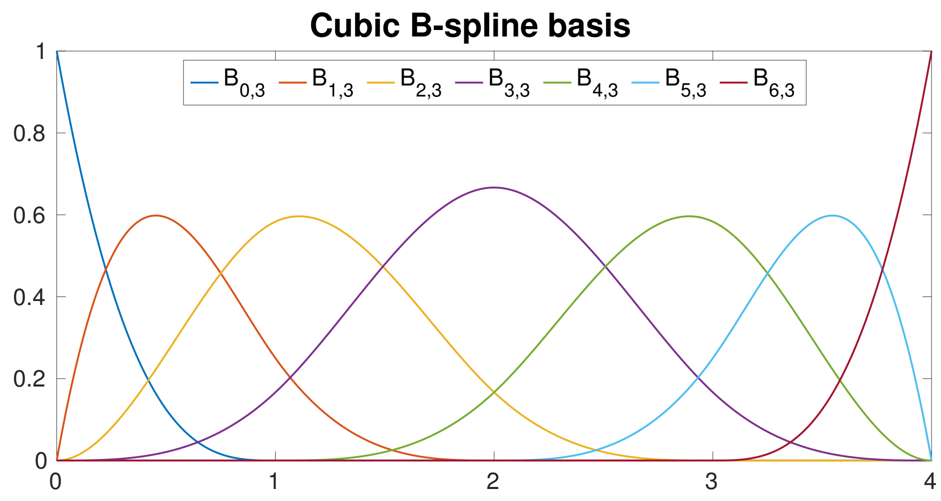


Figure 1.1.3. Cubic B-spline basis defined by an open knot vector $\Xi = \{0, 0, 0, 0, 1, 2, 3, 4, 4, 4, 4\}$.

We can generalize the computation of the kth derivative in the following way

$$B^{(k)}_{i,p}(u) = \frac{p!}{(p-k)!} \sum_{l=0}^{k} a_{k,l} B_{i+l,p-k}(u) \tag{1.1.8}$$

with

$$a_{0,0} = 1,$$

$$a_{k,0} = \frac{a_{k-1,0}}{u_{i+p-k+1} - u_i},$$

$$a_{k,l} = \frac{a_{k-1,l} - a_{k-1,l-1}}{u_{i+p+l-k+1} - u_{i+l}}, \qquad l = 1, \ldots, k-1,$$

$$a_{k,k} = \frac{-a_{k-1,k-1}}{u_{i+p+1} - u_{i+k}}.$$

Again, if the denominator is zero, then the quotient is defined to be zero.

Moreover, a B-splines can always be written as a linear combination of polynomials of higher degree. For simplicity, let us consider the interval $[0, 1]$, then the ith Bernstein polynomial of degree $p-1$ can be written as

$$B_{i,p-1}(u) = \frac{p-i}{p} B_{i,p}(u) + \frac{i+1}{p} B_{i+1,p}(u). \tag{1.1.9}$$

B-splines curves of degree p are defined using basis functions $\{B_{i,p}\}_{i=0}^{I-1}$ and coefficients $\{\boldsymbol{c}_i\}_{i=0}^{I-1}$, which are called control points, in the following way:

$$\boldsymbol{C}(u) = \sum_{i=0}^{I-1} B_{i,p}(u)\boldsymbol{c}_i \tag{1.1.10}$$

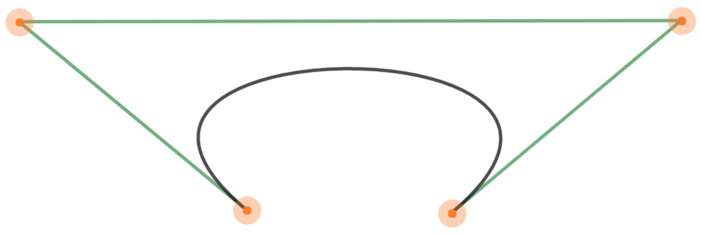


Figure 1.1.4. Cubic B-spline curve. Orange circles correspond to control points green lines indicate the convex hull.

and its corresponding derivative is given by

$$\boldsymbol{J}(u) = \sum_{i=0}^{I-1} B_{i,p}^{'}(u)\boldsymbol{c}_i. \tag{1.1.11}$$

In general, the control points of a curve do not lie on it and if we connect them by straight lines we obtain the so called **control polygon**, which can be considered a good approximation for the curve. Indeed, we can always replace each $B_{i,p}$ in (1.1.10) with basis functions of higher degrees by using (1.1.9), thus we will have more control points. As their number increases, the corresponding control polygon converges to the curve itself. A B-spline curve is always contained in the convex hull of its control polygon (Fig. 1.1.4); to be precise a polynomial segment due to a non-empty knot-span $[u_s, u_{s+1})$ is in the convex hull of the control points $c_{s-p}, \ldots, c_s$. To check the continuity of the curve we need to analyze the continuity of its underlying basis functions, which means that the knot multiplicity and the positions of the control points determine the continuity at knots. If the curve consists of a single polynomial segment, then it is known as Bézier curve and a polynomial segment of a B-spline is called Bézier segment if it can be represented by a Bézier curve.

Rational functions can be represented by associating some weights w_i to the control points

$$c_i^h = (w_i c_i, w_i)^T = (c_i^w, w_i)^T \in \mathbb{R}^{d+1}, \tag{1.1.12}$$

where d is the spatial dimension of the model space. In particular they define a B-spline curve $\boldsymbol{C}^h(u)$ in $\mathbb{R}^{d+1}$. In order to obtain curve $\boldsymbol{C}(u)$ in $\mathbb{R}^d$ we need a perspective mapping $\mathcal{P}$ such that

$$\boldsymbol{C}(u) = \mathcal{P}\left(\boldsymbol{C}^h(u)\right) = \frac{\boldsymbol{C}^w(u)}{w(u)}, \tag{1.1.13}$$

where $\boldsymbol{C}^w(u) = \left(C_1^h, \ldots, C_d^h\right)^T$ are the homogeneous vector components of the curve and the weighting function is given by

$$w(u) = \sum_{i=0}^{I-1} B_{i,p}(u) w_i. \tag{1.1.14}$$

$\boldsymbol{C}(u)$ is called *non-uniform rational B-splines* (NURBS) curve, where the term rational indicates that the resulting curve is a piecewise rational polynomial, whereas non-uniform means that the knots values can be chosen arbitrarily.

The derivative of the NURBS is

$$\boldsymbol{J}(u) = \frac{w(u)\frac{\partial \boldsymbol{C}^w(u)}{\partial u} - \frac{\partial w(u)}{\partial u}\boldsymbol{C}^w(u)}{(w(u))^2}, \tag{1.1.15}$$

with

$$\frac{\partial w(u)}{\partial u} = \sum_{i=0}^{I-1} B^{'}_{i,p}(u) w_i, \tag{1.1.16}$$

$$\frac{\partial \boldsymbol{C}^w(u)}{\partial u} = \sum_{i=0}^{I-1} B^{'}_{i,p}(u) \boldsymbol{c}_i^w. \tag{1.1.17}$$

NURBS become B-splines if all the weights are equal and they have the same properties if the weights are non-negative. Moreover, NURBS curves can be represented as

$$\boldsymbol{C}(u) = \sum_{i=0}^{I-1} R_{i,p}(u) \boldsymbol{c}_i, \tag{1.1.18}$$

with

$$R_{i,p} = \frac{w_i B_{i,p}(u)}{w(u)}. \tag{1.1.19}$$

The weighting function is defined as in equation (1.1.14) and the weights w_i are now associated with B-splines $B_{i,p}$, so the mapping (1.1.18) employs control points $\boldsymbol{c}_i$ of the model space.

We can, also, approximate a function f with a B-spline patch $I_h f = \sum_{i=0}^{I-1} B_{i,p} c_i$. The condition for the interpolation is

$$f(\bar{u}_j) = \sum_{i=0}^{I-1} B_{i,p}(\bar{u}_j) c_i, \quad j = 0, \ldots, I-1. \tag{1.1.20}$$

In this way we can find the unknown coefficients c_i, and we can impose

$$A_u[j,i] = B_{i,p}(\bar{u}_j), \quad i, j = 0, \ldots, I-1, \tag{1.1.21}$$

where A_u is called the *spline collocation matrix.* According to the Schoenberg-Whitney theorem [5, 6]

$$A_u \text{ is invertible} \iff B_{i,p}(\bar{u}_i) \neq 0, \quad i = 0, \ldots, I-1. \tag{1.1.22}$$

Its condition number increases if either $\bar{\boldsymbol{u}}$ approaches the limits of its allowed range or if it is not collocated uniformly. In particular, it gets arbitrary large if two interpolation points approach each other, whereas the others remain fixed. To avoid this problem, it is suggested in [5, 7, 8] to interpolate the function at the Greville abscissae $\boldsymbol{u}^g$ which are

$$u_i^g = \frac{u_{i+1} + u_{i+2} + \ldots + u_{i+p}}{p}, \quad i = 0, \ldots, I-1. \tag{1.1.23}$$

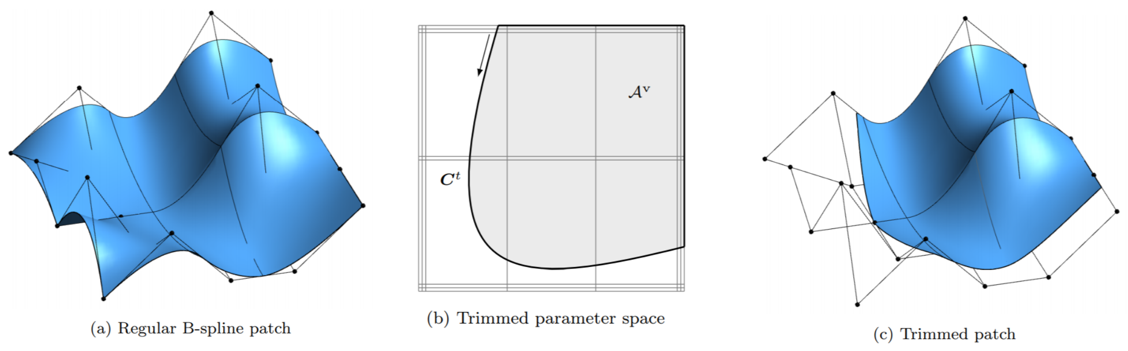

(a) Regular B-spline patch
$\mathcal{A}^{\mathrm{v}}$
$\boldsymbol{C}^t$
(b) Trimmed parameter space
(c) Trimmed patch

Figure 1.2.1. Trimmed tensor product surface: (a) regular surface defined by a tensor product basis, (b) trimmed curved and corresponding visible area $\mathcal{A}^{\mathrm{V}}$ given by the direction of $\boldsymbol{C}^t$, (c) resulting trimmed surface. Image taken from article [4].

1.2 Trimming

A trimmed surface is used to represent an arbitrary surface in space. Indeed, surfaces are cut by a trimming or cutting curve $\boldsymbol{C}^t(\tilde{u})$, which is usually a B-spline or a NURBS curve. It can be expressed as

$$\boldsymbol{C}^t(\tilde{u}) = \begin{bmatrix} u(\tilde{u}) \\ v(\tilde{u}) \end{bmatrix} = \sum_{i=0}^{I-1} R_{i,p}(\tilde{u})\boldsymbol{c}_i^t, \tag{1.2.1}$$

where $\boldsymbol{c}_i^t \in \mathbb{R}^2$ are the control points of the trimming curve in the parameter space of the trimmed surface. Connecting trimming curves, we form loops that can include the original boundary of the trimmed patch if it is intersected by the curve itself. These loops have a direction that make us understand which is the visible area, called $\mathcal{A}^{\mathrm{V}}$. An example of trimmed patches is shown in Fig. 1.2.1. It must be noticed that the mathematical models to construct the original patch and the related control grid do not change and they are not updated to reflect the trimmed boundary. This means that these curves are an useful tool to represent arbitrary surfaces and visible areas, but they do not solve other problems, such as a smooth connection of two adjacent patches along a trimming curve.

Now, we are going to present the problem of approximating the solution partial differential equations on trimmed domains.

1.2.1 PDE problems on trimmed domains

Let us consider an elliptic partial differential equation posed on a domain Ω, then Ω is called a *non-conforming multipatch trivariate volume* (nCMTV) [34] if it is constructed as a collection of patches (*trivariate* in $\mathbb{R}^3$), $T_1, \ldots, T_k$. For each T_l we have a parametrization $\boldsymbol{F}_l : \hat{T} \mapsto T_l$, $l = 1, \ldots, k$, from the parameter space to the current element, which are, in general, different for every edge or face. If the parametrizations are the same the domain Ω is called *conforming multipatch trivariate volume* (CMTV)

The computational domain is defined as

$$\Omega = \Omega_0 \setminus \bigcup_{i=0}^{N} \bar{\Omega}_i, \tag{1.2.2}$$

where $\Omega_1, \ldots, \Omega_N$ are nCMTV and Ω_0 can be meshed by conforming meshes. Thus, the boundary of Ω is composed by two parts: a non trimmed part $\partial\Omega \cap \partial\Omega_0$ and a trimmed boundary $\partial\Omega \setminus \partial\Omega_0$. We indicate with $\Gamma_D \subset \partial\Omega \cap \partial\Omega_0$ a connected non-empty subset; this means that we can impose essential boundary conditions only on the non-trimmed part of the boundary; if this is not the case the problem requires special care [30, 31, 32]. Consequently we have $\Gamma_N = \partial\Omega \setminus \bar{\Gamma}_D$ and we define

$$V = \left\{ u \in H(\Omega)^k : u|_{\Gamma_D} = 0 \right\}, \tag{1.2.3}$$

where $H^s(\Omega)$ denotes the Sobolev space of order $s \geq 0$, cf. [35], and where $k = 1$ when we deal with diffusion problems and $k = 3$ for elasticity problems. We equip V with the natural norm induced by the inner product and denoted by $\|\cdot\|_V$. We assume to have a bilinear form $a(\cdot, \cdot)$ acting on $V \times V$ which is

- continuous, i.e. $\exists M > 0$ such that $a(u, v) \leq M \|u\|_V \|v\|_V$;
- coercive, i.e. $\exists \alpha > 0$ such that $a(u, u) \geq \alpha \|u\|_V^2$.

Therefore, if the data f and g are sufficiently regular and $\Gamma_D \neq \emptyset$, the following problem

$$\text{Find } u \in V : a(u, v) = \int_\Omega fv + \int_{\Gamma_N} gv \quad \forall v \in V \tag{1.2.4}$$

has a unique solution, thanks to Lax-Milgram theorem [40] and (1.2.4) is called weak form.

In order to solve the corresponding numerical problem, Ω_0 has been constructed in such a way that it is described by n_0 patches $T_{0,1}, \ldots, T_{0,n_0}$ and n_0 parameterizations $\boldsymbol{F}_{0,1}, \ldots, \boldsymbol{F}_{0,n_0}$, which let us transform the parameter space $\hat{T}$ into the physical domain. On each trivariate of Ω_0, we have a collection of B-splines

$$\mathbb{B}_{0,j} = \{B_{0,j,k} = \hat{B}_{0,j,k} \circ \boldsymbol{F}_{0,j}^{-1}, \quad k \in \mathcal{I}_{0,j}\}, \tag{1.2.5}$$

where $\mathcal{I}_{0,j}$ is the index of the B-spline function on the patch $T_{0,j}$.

If $h = \max_j h_j$ and h_j is the diameter of the largest knot span in $T_{0,j}$, we can define a mesh $\mathcal{T}_h(\Omega_)$, composed by a collection of elements $Q \in \mathcal{T}_h(\Omega)$. Among them, we can distinguish between two families:

- $\mathcal{T}_h^{\text{untr}}(\Omega)$ is the collection of $\mathcal{Q} \in \mathcal{T}_h(\Omega)$ such that $\mathcal{Q} \cap \Omega = \mathcal{Q}$, which means that they are the non trimmed elements;
- $\mathcal{T}_h^{\text{trim}}(\Omega)$ is the collection of $\mathcal{Q} \in \mathcal{T}_h(\Omega)$ such that $\mathcal{Q} \cap \Omega \neq \mathcal{Q}$, so they are the trimmed elements.

To find the approximated solution of a PDE, we need to compute integrals in all the elements of the mesh $\mathcal{T}_h(\Omega)$, then the final result will be given by the sum of all the contributes, i.e. $\int_\Omega \xi = \sum_{\mathcal{Q}\in\Omega} \int_{\mathcal{Q}} \xi$. Moreover, we can separate the integral into two terms as follow

$$\int_\Omega \xi = \sum_{\mathcal{Q}\in\mathcal{T}_h^{\text{untr}}(\Omega)} \int_{\mathcal{Q}} \xi + \sum_{\mathcal{Q}\in\mathcal{T}_h^{\text{trim}}(\Omega)} \int_{\mathcal{Q}\cap\Omega} \xi. \tag{1.2.6}$$

If the element is not cut, i.e. $\mathcal{Q} \in \mathcal{T}_h^{\text{untr}}(\Omega)$, then the integral is computed in a standard way, e.g. using Gauss-Legendre quadrature rule. Otherwise, if $\mathcal{Q} \in \mathcal{T}_h^{\text{trim}}(\Omega)$ we need to find a procedure to compute the integral over the active part $Q = \mathcal{Q} \cap \Omega$ of the cut Bézier elements. First of all we need to identify cut elements and we show how to do it in the following subsection, then we will dedicate an entire section to the procedures that can be used to perform integration.

1.2.2 Element Detection or Slicing

Element detection is a local technique used to determine if an element must be treated with particular care. Indeed we can have:

- **Exterior elements**, whose knot spans are completely out of the domain of interest and that can be ignored;
- **Interior elements**, whose knot spans are completely contained in the domain of interest and that can be treated as in "classical" isogeometric analysis;
- **Cut elements**, which contain the trimmed curve and must be treated separately.

In Fig. 1.2.2 there are the most common cutting patterns, e.g. [10, 11, 12, 13, 14], but it is important to note that they are not the only possible cases; for instance in a single element, there could be more than one trimming curve, especially if the grid is coarse.

We define the type of each cut element by its number of edges, so if we consider the patterns in Fig. 1.2.2 we can have type 2, 3, 4, 5 or 6. For what concerns interior and exterior elements, by convention, we indicate them as of type 1 and -1, respectively (Fig. 1.2.3). The advantage of this local approach is that the complexity of the trimming curve is reduced when it is restricted to a single element and so it is easier to deal with it. Usually the more refined the grid is, the less complex the trimming curve is; this is why local refinement is a common way to resolve invalid cutting patterns, as in Fig. 1.2.3. This refinement is particularly useful when it comes to integration, indeed no new knots are introduced, but the invalid element is subdivided into several valid integration regions.

Several algorithms to assign types have been proposed in literature; we will present the one by Kim et al. [10, 11] and the one by Schmidt et al. [14]. The procedure suggested by Kim is shown in Fig. 1.2.4. First of all we need to identify interior and exterior elements; to do this, we compute the minimal signed distance $d_{i,j}$ from the center of the element to the curve for each non-zero knot-span. Then we identify cut elements, comparing this distance with the radii $r_{i,j}^{\text{in}}$ and $r_{i,j}^{\text{out}}$ of the inscribed and circumscribed circles of the element (Fig. 1.2.4(a)). Of course if $r_{i,j}^{\text{in}} < |d|$ then we can identify it as a cut element. Otherwise if $r_{i,j}^{\text{in}} \leq |d_{i,j}| < r_{i,j}^{\text{out}}$ we need to compute also the signed distance

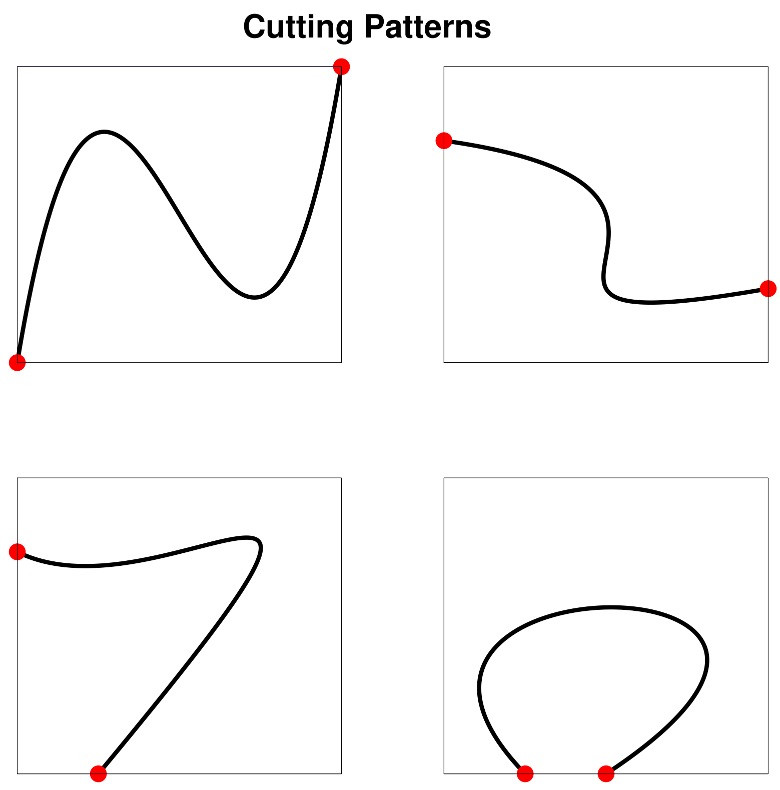


Figure 1.2.2. Most common valid cutting patterns. Red dots indicate the intersection between the trimming curve and the element. The visible area could be both the one under and the one above the curve.

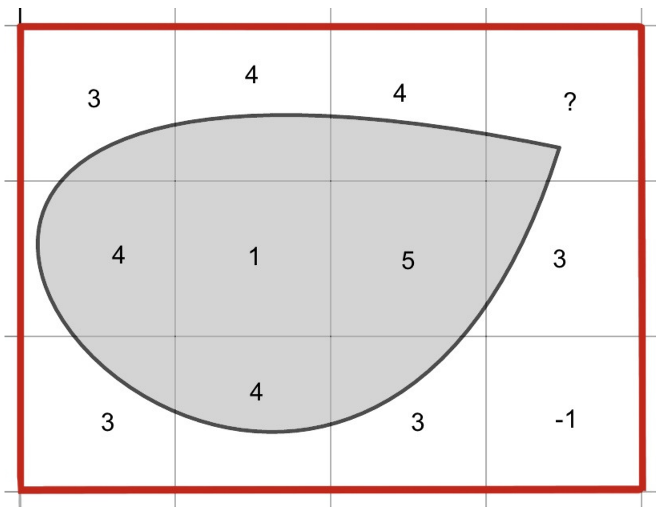


Figure 1.2.3. Trimmered parameter space with corresponding type labels: 1 indicate internal elements, -1 external ones, trimmed elements are referred to using their number of edges, i.e. 3, 4 or 5. The question mark indicates the invalid case.

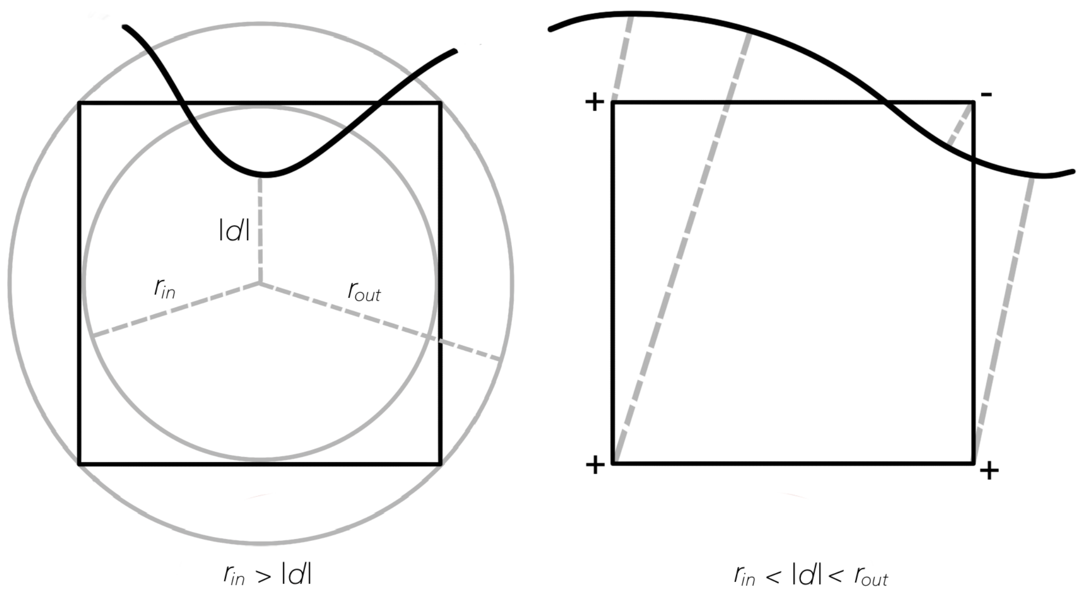


Figure 1.2.4. Kim's procedure [10, 11] to detect cut elements: (right) first assessment based on the inscribed and circumscribed circles of the element; (left) if necessary, further comparison of the signed distance of element corners.

of the element corner nodes to the trimming curve (Fig. 1.2.4(b)). Finally, for each cut element, we compute the intersections of the trimming curve with the grid.

On the other hand, Schmidt's algorithm [14] first denotes all non-zero knot spans as interior elements, then it computes the intersections with the grid and orders them in a non-decreasing way with respect to $\tilde{u}^+$. Then, for each cut element it assigns the right type and detect exterior ones based on their position relative to the cut. The starting point of this procedure is described in Fig. 1.2.5.

Both algorithms require an efficient method to identify the intersections of the parameter grid with the trimming curve, together with the corresponding parametric values $\tilde{u}^+$; in order to do this we can use the Bézier clipping technique, which has been introduced by Sederberg and Nishita [9]. Let us suppose we want to identify the intersections between the curve and the ray l, defined as

$$ax + by + c = 0 \quad \text{with} \quad a^2 + b^2 = 1. \tag{1.2.7}$$

Then we can compute the distance between a point on the curve and the ray as

$$d(u) = \sum_{i=0}^{p} d_i B_{i,p} \quad \text{with} \quad d_i = ax_i + by_i + c, \tag{1.2.8}$$

where d_i are the distances of the control points c_i of the Bézier curve to the ray and $B_{i,p}$ are Bernstein polynomials of degree p. This means that equation (1.2.8) can be seen as a non-parametric Bézier curve $\tilde{\boldsymbol{C}}(u, d(u))$, with d_i related to the corresponding Greville abscissae $u_i = \frac{i}{p}$; this relationship can be seen in Fig. 1.2.6. If we compute the roots of $\tilde{\boldsymbol{C}}(u, d(u))$ we obtain the intersections of $\boldsymbol{C}(u)$ and l. Exploiting the convex hull property of Bézier curves we can select which part of the domain contains the

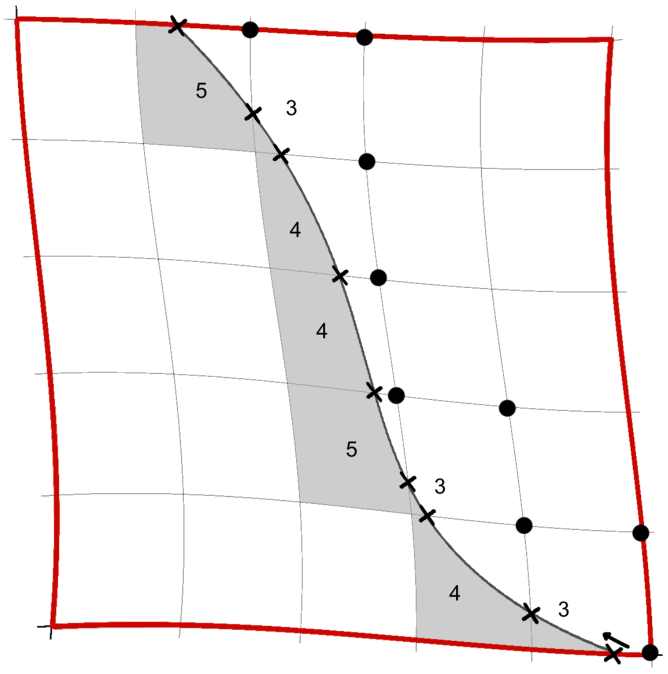


Figure 1.2.5. Starting point for Schmidt's [14] edge detection procedure [14]. White knot spans are not classified yet. Circles are external nodes and crosses indicate intersections with the grid. The arrow indicate the direction of the trimming curve.

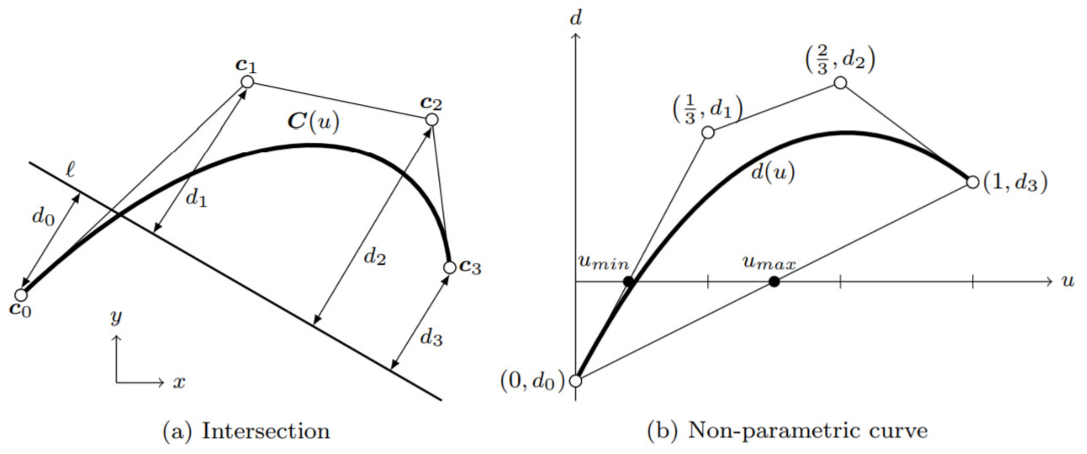


Figure 1.2.6. Bézier clipping: (a) Intersection of $\boldsymbol{C}(u)$ with a ray and (b) the corresponding non-parametric curve used to determine the interval $[u_{\text{min}}, u_{\text{max}}]$ which contains the intersection with the ray. Image taken from article [4].

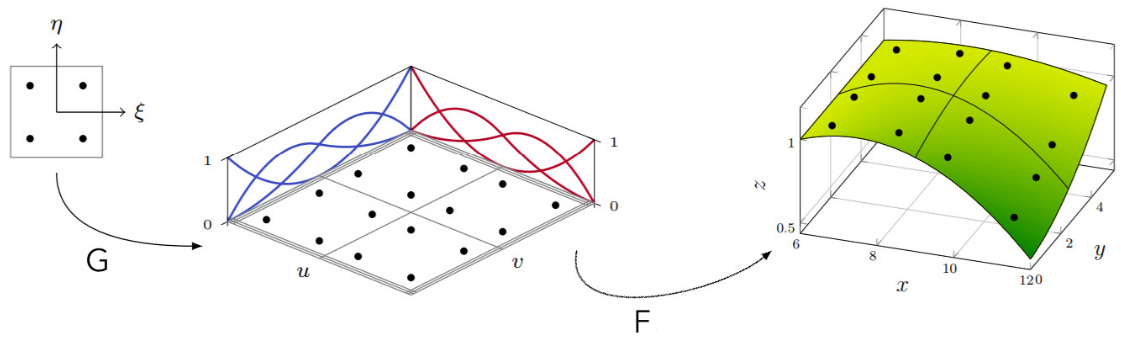


Figure 1.3.1. Distribution of quadrature points over a regular patch. Image taken from article [4].

intersection, erase the other ones and repeat the procedure until a certain tolerance is reached. In particular, if we consider Fig. 1.2.6, the domain is split in three parts by the line that joins the first and the last point of the curve, but only $[u_{\min}, u_{\max}]$ contains the intersection. The curve in this region is extracted and the procedure is repeated.

1.3 Integration

In this section we will present several strategies for integration. Gauss-Legendre quadrature formulae [40] are the most common ones and consist in replacing the integral by a weighted sum of function evaluations at quadrature nodes. Usually, this formula is performed on the reference element $\tilde{Q} = [-1, 1]^2$. This means that we need to determine the point for function evaluation by using a transformation $\boldsymbol{G}(\xi, \eta) : \mathbb{R}^2 \mapsto \mathbb{R}^2$, from the reference element to the parametric one $\hat{Q}$, and the geometrical mapping $\boldsymbol{F}(u, v) : \mathbb{R}^2 \mapsto \mathbb{R}^3$, i.e. $y_g = \boldsymbol{F}(u_g, v_g) = \boldsymbol{F}(\boldsymbol{G}(\xi_g, \eta_g))$ from the parametric to the physical, as in Fig. 1.3.1. Then the formula to compute the integral is given by

$$I_{\hat{Q}} = \int_{\hat{Q}} f(\boldsymbol{x}) d\Omega_{\tilde{Q}} \approx \sum_{g=1}^{n} f(\boldsymbol{y}_g) G_r(u_g, v_g) J_{\tilde{Q}}(\xi_g, \eta_g) w_g. \tag{1.3.1}$$

where $G_r(u, v)$ and $J_{\tilde{Q}}(\xi, \eta)$ keep into account the transformation from the reference element to the current one and they are defined as

$$G_r(u, v) = \sqrt{\det \left(\boldsymbol{J}_F^{\mathrm{T}}(u, v) \boldsymbol{J}_F(u, v)\right)}, \tag{1.3.2}$$

where $\boldsymbol{J}_F$ is the Jacobian matrix of the geometrical mapping and

$$J_{\tilde{Q}}\left(\xi_g, \eta_g\right) = \det\left(\boldsymbol{J}\left(\xi_g, \eta_g\right)\right) \tag{1.3.3}$$

is the Jacobian determinant of $\boldsymbol{G}$ and it is evaluated with respect to the reference coordinates ξ_g and η_g of the integration point $\boldsymbol{y}_g$. In case of regular elements the definition of $\boldsymbol{G}$ is straightforward, but cut elements are more complex and they require special attention. Numerical integration on cut elements may be performed in several

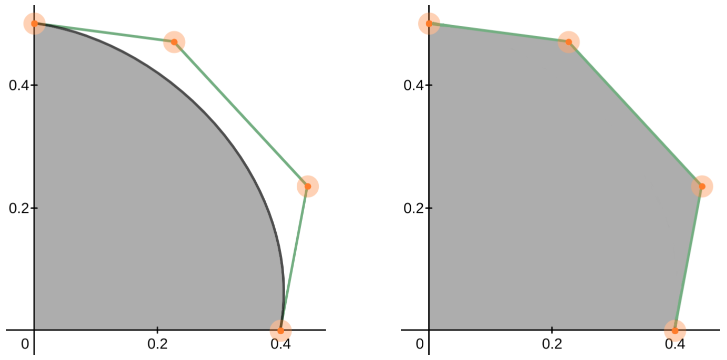


Figure 1.3.2. Trimming curve approximated by its control polygon, in order to create the polytope. Circles indicate the control points.

ways because of different reconstructions of the trimming curve. In the next sections we will present different methods to perform this reconstruction and thus the integration. We will restrict our presentation to the 2D case, but it is relevant to remark that in 3D the computations we are going to describe are much more complicated (see [30]).

1.3.1 Tailored Integration

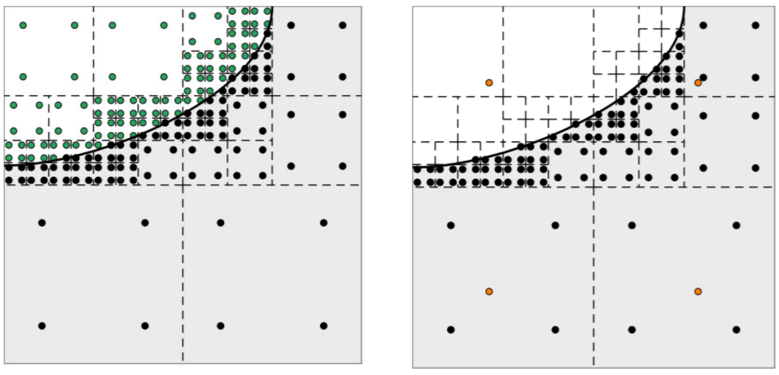
The first scheme we are going to show is called *tailored integration* [15, 16, 17] and the trimming curve is approximated by its control polygon, in order to represent $\hat{Q}$ by a polytope $\hat{\rho}$ as shown in Fig. 1.3.2. Using Lasserre's theorem [18] we can transform the integrals over the polytope $\hat{Q}$ in integrals over its edges, but $\Omega_{\hat{\rho}}$ must be convex. If this is not the case we need to subdivide it in a combinations of convex regions. The line integrals on the right hand side provide reference solutions

$$\sum_{i=1}^{m} f_j(u_i, v_i) w_i = \int_{\Omega_{\hat{\rho}}} f_j(u, v) d\Omega_{\hat{\rho}}, \quad j = 1, \ldots, n, \tag{1.3.4}$$

that are used to compute a suitable quadrature rule for all functions f_j of the desired function space. The integral in (1.3.4) is first computed using a large number of integration points $\boldsymbol{y}_i = (u_i, v_i)$, which is gradually reduced such that the error in (1.3.4) remains below a certain tolerance. Of course, the goal is to minimize the number of integration points. This method is efficient because all the cutting patterns are treated using the same technique, but it involves a very accurate preprocessing phase due to the fact that every cut element must be treated individually. Moreover, an approximation error is introduced when the cut curve is replaced by its control polygon, even if it can be reduced thanks to the property that the control polygon converges to the curve itself.

1.3.2 Adaptive Subdivision

Adaptive subdivision is often used combined with the finite cell method [19, 20, 21, 22]. The idea is to construct quadtrees in 2D and octrees in 3D, which means that each cut element $\hat{Q}$ in the parametric domain is subdivided into 4 axis-aligned cells $\hat{Q}^{\boxplus}$, which are recursively refined into other 4 if they are cut by the curve until a maximal depth is



(a) Conventional (b) Reduced

Figure 1.3.3. Adaptive subdivision, sub-cells indicated with dotted lines : (a) conventional approach, with black points in the valid domain and green points in the exterior one, and (b) reduced approach, with orange points on the whole domain and black points in the valid one. Image taken from article [4].

reached, as shown in Fig. 1.3.3(a). Then, if we identify with I^{c} the integral over the original domain, we obtain

$$I^{\text{c}} = \sum_{i=1}^{I} I^{\text{v}}_{\hat{Q}_i^{\boxplus}}\left(\alpha^{\text{v}}\right) + \sum_{j=1}^{J} I^{-}_{\hat{Q}_j^{\boxplus}}\left(\alpha^{-}\right), \tag{1.3.5}$$

where I^{v} and I^{-} are the integrals over the valid domain $\mathcal{A}^{\text{V}}$ and the complementary exterior domain $\mathcal{A}^{-}$, respectively. Integration points in the valid domain are multiplied by $\alpha^{\text{v}} = 1$, whereas points in the exterior are multiplied by a factor that is almost zero, e.g. $\alpha^{-} = 10^{-4}$, as suggested in [21]. A reduced approach is shown in Fig. 1.3.3(b) and it can be written as follow

$$I^{\text{c}} = \sum_{i=1}^{I} I^{\text{v}}_{\hat{Q}_i^{\boxplus}}\left(\alpha^{\text{v}} - \alpha^{-}\right) + I^{-}_{\hat{Q}}\left(\alpha^{-}\right), \tag{1.3.6}$$

where $I^{-}_{\hat{Q}}\left(\alpha^{-}\right)$ represents the integral over the whole cut element, ignoring the cutting curve. The integration over the valid domain is computed as before, but it has a different weighting factor, i.e. $\left(\alpha^{\text{v}} - \alpha^{-}\right)$. As in the previous approach, every cutting pattern can be addressed with a single algorithm and it is very easy to be implemented also in higher dimensions. Again, an approximation error is introduced, but this time the sub-cells do not necessarily converge to the curve.

1.3.3 Reconstruction of the Trimming Curve with Bézier functions

In this section we are going to show how to use B-splines to represent the trimming curve and place quadrature points in order to create suitable integration formulae.

First of all, for each cut element, we need to find a map $\boldsymbol{F}$ that transforms the parametric domain $\hat{Q}$ into the physical element Q. In principle, the cutting curve on $\hat{Q}$

could be any kind of function, so we decide to approximate it with a B-spline of order $p \geq 1$ (in our numerical results we will consider $p = 3$); the analysis of the optimality of this choice can be found in [34].

The process of reconstruction is very laborious. Indeed when the physical domain Ω is cut, the trimming curve is sampled with a large number of points that are joined by segments in order to create a broken line. Then each cut element Q is transformed into $\hat{Q}$ and all the points of the broken line in that element are mapped as well. At this point a parameterization of the line is created and the points that correspond to the parameterization variable $\tilde{u}^+ = \{0, 1/3, 2/3, 1\}$ are found on $\hat{Q}$. Having the coordinates of four points, we can find the lagrangian polynomial of degree 3 that interpolates them. In 3D, finding appropriate interpolation points is far more complicated and implies the creation of a grid with Gmsh [36] and Open CASCADE [33]. We will not show the details on the reconstruction in three dimensions but we refer the reader to [30]. Then with a change of basis we obtain the control points of the corresponding Bézier representation.

After that, we look for a map $\boldsymbol{G}$ to transform the reference element $\tilde{Q} = [0, 1] \times [0, 1]$ into the parametric one. This last procedure can be directly applied to elements of type 3 and 4, as in Fig. 1.3.4. Type 5 and 6 elements are divided into elements of type 3 and 4, as in Fig. 1.3.5.

The resulting map from the reference to the physical domain is

$$\mathcal{X} = \boldsymbol{F}(\boldsymbol{G}(\xi, \eta)). \tag{1.3.7}$$

Essentially we place into the reference element suitable quadrature points, i.e. we use the Gauss Legendre rule, and then we pull them back into the parametric domain through $\boldsymbol{F}$. Using the map $\boldsymbol{G}$ we can transform integrals from the physical domain to the parametric as:

$$\int_Q \xi = \int_{\hat{Q}} \xi \circ \boldsymbol{G}^{-1} \tag{1.3.8}$$

At this point we can apply the integration formula with the transformed quadrature points and weights.

1.4 Conclusions

As it has been shown in previous sections the process to compute integrals over trimmed surfaces requires many geometrical manipulations, that are sometimes rather unstable. First of all, we need to identify trimmed elements, then we compute a re-parameterization of cut Bézier in order to build suitable integration formulae. In the end, we need to pull back quadrature points from the reference element to the parametric one. In 3D this is even more complicated, as pointed out in [34], and the problem still has open questions. In particular, computer-aided design (CAD) geometric descriptions represent only the boundary and not the interior and they allow for boolean operations, such as union or subtraction, among spline surfaces and primitives. This means that it is not easy to derive a valid mesh and so the conversion to a finite element (FE) domain remains an open problem. Thus a new method has been recently introduced by Elber et al. [29]

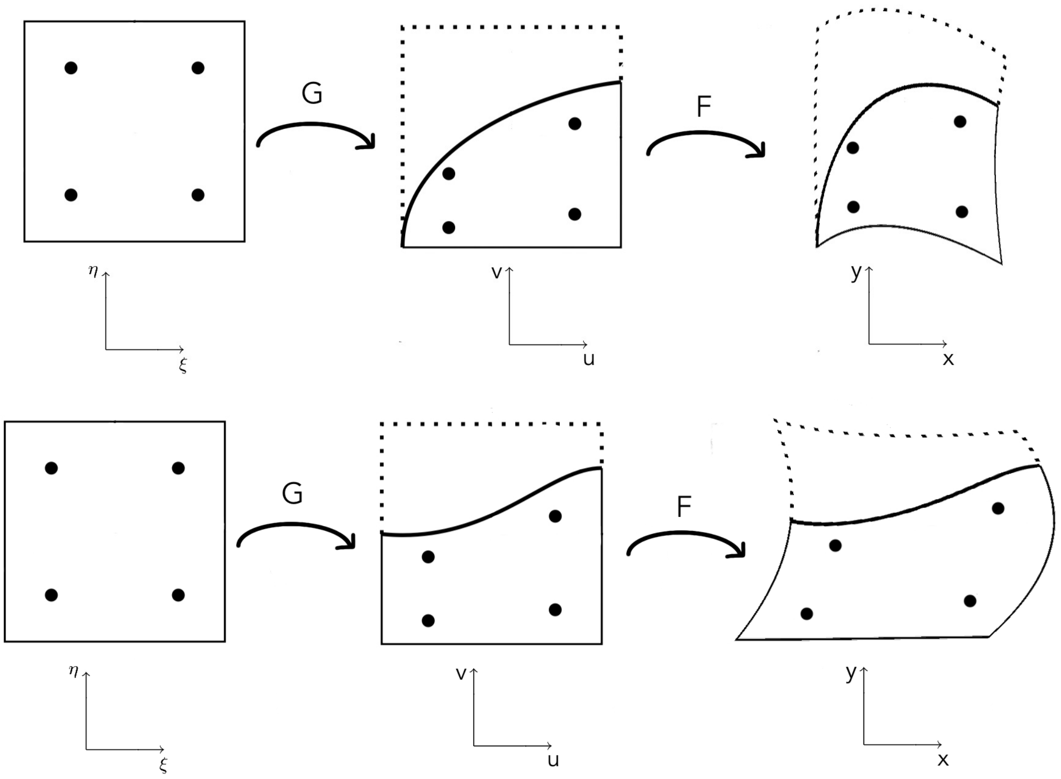


Figure 1.3.4. For elements of types 3 and 4 we can find a transformation $\boldsymbol{G}$ to map the quadrature points from the reference element to the one in the parametric space. Then, we look for a transformation $\boldsymbol{F}$ to pass from the parametric to the physical space. In the parametric domain the cutting curve is approximated with a Bézier curve of degree p.

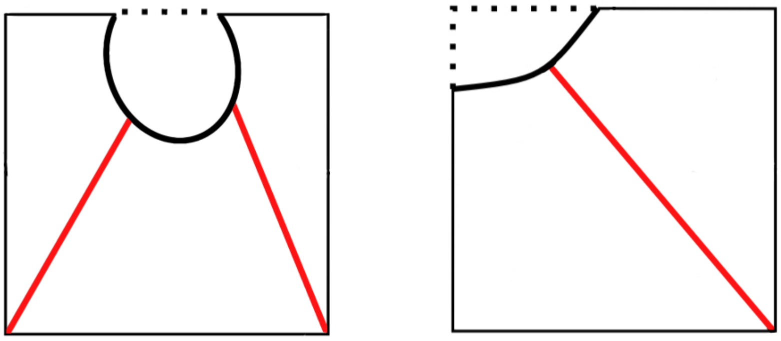


Figure 1.3.5. On the left we have an element of type 6. It is divided into two elements of type 3 and one element of type 4. On the right we have an element of type 5 which is divided into two elements of type 4. In this way we can find for each new element the transformation $\boldsymbol{G}$ from the reference element.

and it allows to see geometries as volumetric representations, which means they are described through the geometric representation of the volume they occupy. For further details we refer the reader to [34].

In the following we propose an approach to find a way to reduce the geometrical manipulations required to compute those integrals. A promising idea is to use the new techniques introduced by machine learning. In particular we decide to represent cut elements through images. Every pixel will have value $+1$ if it contains the cutting curve or the visible boundary. The process to create these images is simple and straightforward even in 3D and it could simplify the computations.

Chapter 2

Neural Networks

In this chapter we introduce the main tools from the huge and fast increasing field of neural network (NN) that we are going to use in the design of our NN based integration rules on trimmed Bézier elements. In particular, we will present Machine Learning (ML) and the tasks for which it is used in Section 2.1. Then we will provide an overview on Feed Forward Neural Networks in Section 2.2 and on Convolutional Neural Networks in Section 2.3.

2.1 Machine Learning and Neural Networks

Machine learning [24] is the instrument we use to find inherent rules or dependencies from a large amount of data, not intuitively or by expert's insight but automatically. The aim is to create a model that can be used for different tasks. The most common ones are:

- **Classification**: given an input, the model must be able to recognize to which of k categories it belongs to. The ways to achieve this task are several. For instance, the model could train a function $f : \mathbb{R}^n \mapsto \{1, \ldots, k\}$ and the output $y = f(\boldsymbol{x})$ for a generic input vector $\boldsymbol{x}$ is a number that identifies the correct category. An alternative could be to have an output that represents the probability a specific input belongs to each of the k classes. An example of the use of the classification task is object recognition, where the input is an image, for instance the MNIST Database of handwritten digits [25], and the algorithm must identify the object shown in the image itself. This process can be *supervised* or *unsupervised*; the former means the network knows the output and it tries to adjust the weights in order to make the computed output as close as possible to the desired one, the latter means the network doesn't know the correct results we want to obtain and it is left to itself to find internal rules that can or can not lead to a stable state in some number of iterations (*epochs*).

- **Regression**: in this case the model must be able to predict a value given an input. More precisely, it has to learn a function $f : \mathbb{R}^n \mapsto \mathbb{R}$, which is similar to the previous case except for the format of the output. A simple example for this task is linear regression, where our output is given by $y = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$, where $\boldsymbol{x}$ is the input vector and $\boldsymbol{w} \in \mathbb{R}^n$ is a set of weights that determine how each feature x_i affects the prediction. In particular if w_i is positive, then increasing the value of the corresponding feature means increasing the value of the prediction; on the other hand a negative weight means that increasing x_i makes the prediction decrease. If the weight is zero then the feature is not meaningful for the predicted value.

There are several aspects that must be taken into account when using machine learning:

- Selection of the appropriate algorithm with correct input (training set and labels, or teacher signal), output and hyperparameters;
- Cost of the preparation of the data we will use to train the model;
- Computational cost to train the model and acquire appropriate rules;
- Computational cost to apply those rules for classification or regression on new data;
- Accuracy of the classification or the regression for unseen data.

If we consider a generic dataset, it is usually split into two parts. The portion used to train the model is called *train set* and it is used to verify it is not *underfitting*, the one used to assess the performance of the model on unknown data is called *test set* and it is used to verify it is not *overfitting*. Underfitting occurs when the model is not able to obtain a sufficient low error on the training set. Overfitting is "the production of an analysis that corresponds too closely or exactly to a particular set of data, and may therefore fail to fit additional data or predict future observations reliably" [23]. This means that is performs very well on the training set, but it does not for unseen data. An example of underfitting and overfitting can be seen in Fig. 2.1.1.

Dividing the dataset into two portions can be problematic when it is too small, so we can use the so called *k-fold cross validation*. This procedure divides the dataset into k subsets of equal size and uses $k-1$ subsets to train the model and the remaining one to test it. Then it iterates this process by choosing another subset as testing set until all subsets are used as test once.

If we want to assess hyperparameters and choose the ones that give the best performance we can use a *validation set*, which means that we train our model with different hyperparameters, we assess the performance of each model created on the validation set, we choose the ones with the best accuracy and with that values we train again our model. The evaluation for unseen data is always done on the test set.

2.2 Feed Forward Neural Networks

Artificial neural networks (ANNs) [24] are networks of simple processing elements, operating on their local data and communicating with other elements. These elements are called 'neurons' because the way in which the networks works reminds of the structure of real brain, in which neurons receive signals that are processed and send output signals to other units. The connections between neurons are marked by real numbers, called weight coefficients; the higher the weight is the more important the connection between the two neurons is considered.

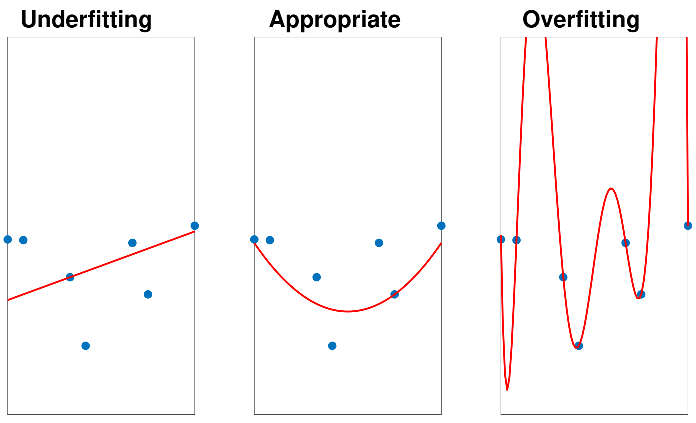


Figure 2.1.1. (Left) The model is underfitting the data, indeed it cannot capture its curvature. (Center) A quadratic function fit to the data generalizes well to unseen points. (Right) A polynomial of high degree fit to the data suffers from overfitting. Indeed, even if it passes exactly through all the training points, it has a deep valley between two points that does not appear to describe the underlying function.

In principle, a neural network (NN) has the power of an universal approximator, i.e. it can realize an arbitrary mapping of one vector space onto another vector space. The process of 'capturing' the unknown relationships between data is called *training of neural networks* and the aim of the model is to predict in the best possible way unseen data. An example of a feed-forward neural network is shown in Fig. 2.2.1, which consists in multiple layers with fundamental processing components called units. A neural network composed by N layers, where $N \geq 1$ is called depth, means that is has a first layer, which is the input layer, $N-1$ hidden layers and a last layer which is the output. In a standard fully-connected feed-forward NN each unit has a weighted connection with every neuron in the neighboring layers, and there are no connections between units on the same layer or in non-neighboring ones. The input is processed sequentially from the input layer to the output via connections between neurons. If n_I represents the number of input units and n_O one of output ones, then the neural network can be seen as a map

$$F_{NN} : \mathbb{R}^{n_I} \mapsto \mathbb{R}^{n_O}. \tag{2.2.1}$$

The number of hidden layers and the number of units for each layer (width), are hyperparameters that must be tuned in order to gain the best possible results in terms of accuracy and complexity of the model. To be more precise, a unit is a multiple-input-single-output processing device, whose inputs come from the previous layer and whose output is a single value that has to be passed to the neurons to which it is connected. Each unit in the hidden or output layer processes an input which is a weighted sum of the outputs of the previous layer. The output signal is the result of a function, called activation, for the sum as follow

$$O_j^p = f(U_j^p) = f\left(\sum_{i=1}^{n_{p-1}} w_{i,j}^{p-1} \dot{O}_i^{p-1} + \theta_j^p\right), \tag{2.2.2}$$

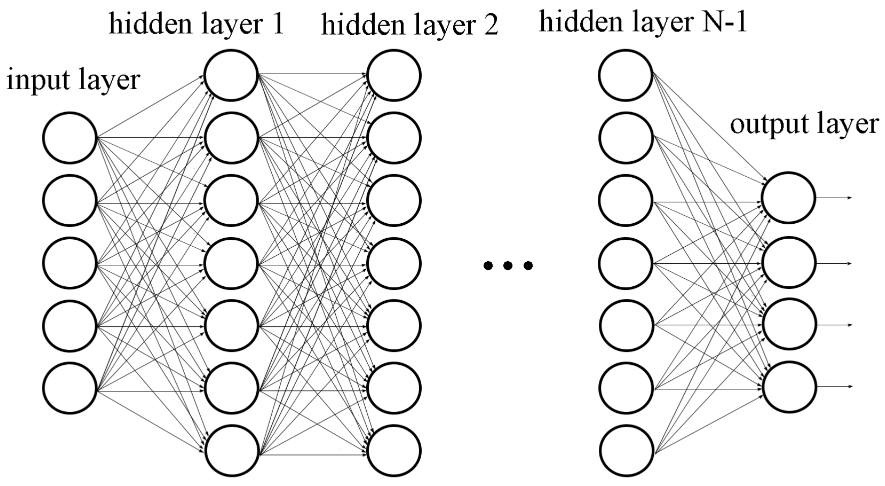


Figure 2.2.1. Feed forward neural network with $N-1$ hidden layers.

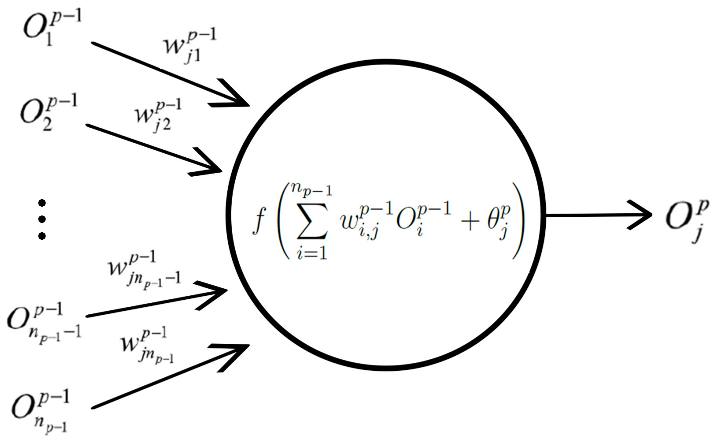


Figure 2.2.2. Schematic diagram of a unit. At each layer every unit sends an output O_j^p to all the units of the following layer, after having processed multiple data given by the units in the precedent layer.

where O_i^{p-1} and $w_{i,j}^{p-1}$ are respectively the output and the connection weight of the ith unit of the $(p-1)$th layer, θ_j^p, U_j^p and O_j^p are respectively the bias, the input and the output of the jth unit in the pth layer, f is the activation function and n_{p-1} is the number of units in the $(p-1)$th layer. A schematic diagram of how a unit works is shown in Fig. 2.2.2. In general, a non-linear, non decreasing function is used as activation; the most common ones are the logistic sigmoid, the hyperbolic tangent and the rectified linear (ReLU) [24].

The sigmoid function is defined as

$$f(z) = \sigma(z) = \frac{1}{1+\exp(-z)}, \tag{2.2.3}$$

whereas the hyperbolic tangent is

$$f(z) = \tanh(z), \tag{2.2.4}$$

but the two functions are linked by the relation

$$\tanh(z) = 2\sigma(2z) - 1. \tag{2.2.5}$$

They saturate to a high value when z is very positive, to a low value when it's very negative and they are only strongly sensitive to their input when it is near 0. This can make gradient-based learning very difficult and it is why they are usually not used in hidden layers. On the other hand, they are applied on the output, when the expected result is bounded.

The ReLU function is defined as

$$f(z) = \max\{0, z\}. \tag{2.2.6}$$

Units with ReLU as activation function are very easy to optimize because they are very similar to linear ones, the only difference is that they are zero if the input variable is negative. The first derivative remains 1 when the unit is active, whereas the second derivative is 0 almost everywhere. This means that the gradient direction is very useful for learning.

A pair of the input data and the corresponding output data, or the teacher signal, is called a *training pattern*, where the aim of the training is to find the rules implicitly inherent in the large amount of training patterns. For the training algorithm the *mean square error* (MSE) is usually adopted. Which means that the weights are chosen in such a way that they minimize

$$\text{MSE}_{\text{train}} = \frac{1}{n} \sum_{i=1}^{n} \left(\hat{\boldsymbol{y}}^{(\text{train})} - \boldsymbol{y}^{(\text{train})} \right)_i^2, \tag{2.2.7}$$

where $\boldsymbol{y}^{(\text{train})}$ are the value predicted by the model and $\hat{\boldsymbol{y}}^{(\text{train})}$ are the teacher signals. One way of measuring the performance of the model is by checking the MSE_{test}, which is computed using the $\boldsymbol{y}^{(\text{test})}$ and $\hat{\boldsymbol{y}}^{(\text{test})}$. Usually, the error defined in Eq. (2.2.7) gradually decreases after every iteration (epoch). Indeed, if not initialized, the model starts its training with random weights and it improves them after every iteration relying on optimizers to reach a minimum.

The function we want to minimize is called *loss* and the optimizers used to find the minimum can be several. In our numerical examples we will use Stochastic Gradient Descend (SGD) [37] or Adam [38]. If we consider sum structured objective functions , such as

$$f(\boldsymbol{x}, \boldsymbol{w}) = \frac{1}{n} \sum_{i=1}^{n} f_i(\boldsymbol{x}, \boldsymbol{w}), \tag{2.2.8}$$

where f_i is the cost function of the $i-$th datapoint and a learning rate γ, then an iteration of SGD is defined as in Algorithm 2.1. On the other hand, Adam algorithm adopts different learning rates using estimates of first and second moments of the gradient. In particular β_1 and β_2 control the decay of the exponential moving average of the gradient and the squared gradient. This scheme is shown in Algorithm 2.2.

There is an *universal approximation theorem* [24] for feed-forward networks with hidden layers which states the following

Algorithm 2.1 SGD learning algorithm; the procedure ends when all training patterns yield an error below a defined threshold.

Input: initial weights $\boldsymbol{w}_0$, cost function $f(\boldsymbol{x}, \boldsymbol{w})$, initial learning rate γ, tolerance ϵ and maximum number of epochs T

Output: final weights $\boldsymbol{w}_{\text{opt}}$

```
1: $t \leftarrow 0$
2: while $t < T$ and $f(\boldsymbol{x}, \boldsymbol{w}_t) < \epsilon$ do
3:     sample $i \in [n]$ uniformly at random
4:     $\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t - \gamma \cdot \nabla_{\boldsymbol{w}} f_i(\boldsymbol{x}, \boldsymbol{w}_t)$
5:     $t \leftarrow t + 1$
6: end while
7: $\boldsymbol{w}_{opt} = \boldsymbol{w}_t$
```

Algorithm 2.2 Adam learning algorithm; the procedure ends when all training patterns yield an error below a defined threshold. Here $\boldsymbol{g}_t^2$ indicates the elementwise square $\boldsymbol{g}_t \odot \boldsymbol{g}_t$. Good choices for the parameters are $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\alpha = 0.001$.

Input: initial weights $\boldsymbol{w}_0$, cost function $f(\boldsymbol{x}, \boldsymbol{w})$, initial learning rate γ, exponential decay rates for the moment estimates β_1, $\beta_2 \in [0, 1)$, tolerance ϵ and maximum number of epochs T

Output: final weights $\boldsymbol{w}_{\text{opt}}$

```
1: $t \leftarrow 0$
2: $\boldsymbol{m}_0 \leftarrow \mathbf{0}$ (Initialize 1st moment vector)
3: $\boldsymbol{v}_0 \leftarrow \mathbf{0}$ (Initialize 2nd moment vector)
4: while $t < T$ and
5:     $t \leftarrow t + 1$ $f(\boldsymbol{x}, \boldsymbol{w}_t) < \epsilon$ do
6:     $\boldsymbol{g}_t \leftarrow \nabla_{\boldsymbol{w}} f(\boldsymbol{x}, \boldsymbol{w}_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
7:     $\boldsymbol{m}_t \leftarrow \beta_1 \cdot \boldsymbol{m}_{t-1} + (1 - \beta_1) \cdot \boldsymbol{g}_t$ (Update biased first moment estimate)
8:     $\boldsymbol{v}_t \leftarrow \beta_2 \cdot \boldsymbol{v}_{t-1} + (1 - \beta_2) \cdot \boldsymbol{g}_t^2$ (Update biased second raw moment estimate)
9:     $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
10:    $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
11:    $\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} - \gamma \cdot \boldsymbol{m}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + 10^{-8})$ (Update parameters)
12: end while
13: $\boldsymbol{w}_{opt} = \boldsymbol{w}_t$
```

Theorem 2.2.1. *of the fuction we want to minimize A linear output layer composed by at least one hidden layer with any "squashing" activation function, such as the logistic sigmoid, can approximate any Borel measurable function from one finite-dimensional space to another with any desired zero amount of error, provided that the network is given enough hidden units.*

It is, also, proved that a feed-forward neural network of more than three layers can simulate any non-linear continuous function with any precision desired [26, 27]. According to LeCun et al. [28], training a neural network with more than three hidden layers is called *deep learning.* Usually, if we want to approximate complex non linear problems in a better way, we can try to add more hidden layers or units, but of course in this way it is more difficult to train the model and it can require more time to gain a stable state. Moreover, in the creation of the model, the computational time to predict an output must be taken into account, and we have to remember that it is $\mathcal{O}(\text{depth} \times (\text{width})^2)$. So we really need to be careful when we choose the number of units for each layer because it could make the computational time explode. Nonetheless, it is possible to reduce the evaluation time by checking the values of the weights after every epoch and strongly impose to the ones below a certain tolerance to be equal to 0. This reduces the number of connections and, consequently, the time for prediction.

However we need to keep in mind that machine learning promises to find rules that are probably correct about most members of the set they deal with; this means that it gives probabilistic and not certain rules. Unfortunately, there exists the so called **no free lunch theorem** [24], which states that:

Theorem 2.2.2. *Averaged over all possible data-generating distributions, every classification algorithm has the same error rate when classifying previously observed points.*

This means it does not exist a machine learning algorithm that is the best for all the possible cases. Of course, this is true only if we average over all possible data-generating distributions. If we make assumptions about the kinds of probability distributions we have in real world applications, then we can design networks that work well on these distributions. The goal of machine learning is not to find an universal learning algorithm, but one that performs well on our specific data.

2.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) [24] are particularly useful when we have input data with a grid-like topology. For instance, images are 2-D grids of pixels with one channel if they are black and white or three channels if they are colored. These networks are called convolutional because they perform a mathematical operation called convolution in at least one of its layers. Formally, if we consider two functions $f(\cdot)$ and $h(\cdot)$, then the **convolution** is defined as

$$s(\boldsymbol{t}) = (f * h)(\boldsymbol{t}) = \int f(\boldsymbol{x})h(\boldsymbol{t} - \boldsymbol{x})d\boldsymbol{x}. \tag{2.3.1}$$

In the context of neural networks convolution is a discrete operation where f is the input I and h is the kernel K, that is usually a multidimensional array of parameters that are adapted by the learning algorithm. We can express the operation in 2.3.1 in a discrete way as

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n); \tag{2.3.2}$$

this is the same as doing

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{2.3.3}$$

because it is commutative.

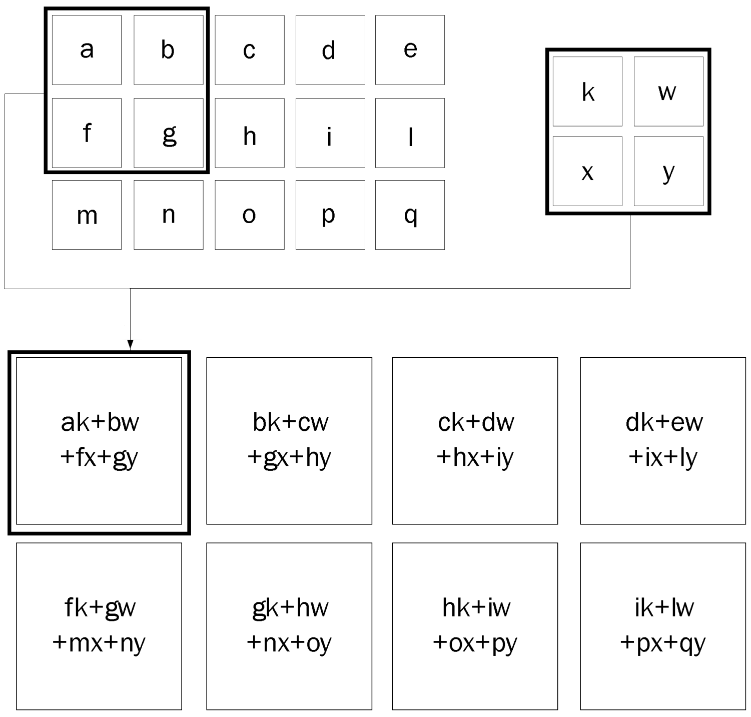


Figure 2.3.1. Example of how a convolutional neural network works with an image composed by pixels and a kernel of size 2×2.

Convolution is particularly useful when we deal with edge detection or sparse input. Indeed, traditional neural networks use matrix multiplication, ending up with a fully connected model, whereas convolutional NNs have sparse interaction, also called sparse connectivity or sparse weights, which is obtained using a kernel that is far smaller than the input. An example is shown in Fig. 2.3.1. This sparse connectivity requires to store fewer parameters and so it is less expensive in terms of memory and statistical efficiency. The improvement is quite significant even in the evaluation time; indeed using a fully connected model with n_I inputs and n_O outputs we have $\mathcal{O}(n_I \times n_O)$ runtime. Whereas if we have sparse connectivity with k as maximum number of connections, then we end up with $\mathcal{O}(k \times n_O)$ runtime, where k is usually taken several orders of magnitude smaller than n_I. Channels are used to extract the main features present in the input and the network is able to reach a local minimum in fewer iterations. Usually after the convolution a pooling is performed; this means that a pooling function is used to replace the output of a cell with a combination of its neighborhood. The most common ones are max pooling, average pooling, the L^2 norm of a rectangular neighborhood or a weighted average based on the distance from the central pixel. These operations are useful to make the representation invariant with respect to small translation of the input.

2.4 Conclusions

Machine Learning and Neural Networks have been introduced in many fields in the last few years and the possibilities they offer are huge. The network is trained by a proper input that is able to span many possible situations. The aim of the model is to generalize in the best possible way unseen data. Of course, the most delicate part is the construction of the training dataset, which is still an art. Even the choice of hyperparameters can affect the results in a significant way, so it is possible that the training phase requires a lot of time. The advantage is that it is done once and for all and the evaluation should be much faster.

Moreover, CNNs will be very useful in the construction of networks to create integration rules on trimmed Bézier elements. Indeed, we will decide to avoid to reconstruct the trimming curve and pass it to the model as an image. Thus, we will divide our domain in pixels and assign values +1 to the ones that represent the cutting curve and the visible boundary. Performing convolution on top of the network let us extract the main features of the curve, i.e. its slopes, and this allows the training to reach a stable state faster.

Chapter 3

Design of NN to compute matrix entries

In this section we are going to show the numerical results we obtained when integrating several functions using neural networks. The aim is to obtain a tool to compute integrals on cut domains. We have used the mean squared error (mse) as metric to minimize the error between the output and the teacher signal (loss function) and Adam [38] as optimization technique, unless otherwise specified.

As this is an almost unexplored field of research, we start to design NN algorithms for simple problems and, progressively, we face more complex situations. In particular, in Section 3.1 we will construct a model to integrate a very simple function in 1D, then in Section 3.2 we will try to predict mass and stiffness matrix entries using as input the control points of the trimming curve. Finally, in Section 3.3 we will predict the same matrix entries using a new approach: we will pass to the model images that show the visible boundary of cut elements and we will exploit CNNs to reach convergence faster.

3.1 The simplest integration problem

First of all we created a model to compute numerically

$$\int_{x_0}^{x_1} x^p dx, \tag{3.1.1}$$

whose exact integral is

$$\int_{x_0}^{x_1} x^p dx = \frac{x_1^{p+1} - x_0^{p+1}}{p+1}. \tag{3.1.2}$$

We took $p \in \{1, \dots, 7\}$, $x_0 \in [0, 0.25]$ and $x_1 \in [0.75, 1]$ to avoid machine errors in the exact computation of (3.1.2). Our training set was composed by 10000 sets of $\{x_0, x_1, p\}$ randomly chosen and our test by 100 points randomly chosen, as well. As labels, we computed the exact value of the integral (3.1.2).

In order to optimize parameters we have created several models with different numbers of hidden layers (depth) and units for each layer (width), specifically we considered depth $= \{2, \dots, 6\}$ and width $= \{50, 100, 150, 200, 250, 300\}$. As activation we have imposed the ReLU function for the hidden layers and the sigmoid for the output; indeed when $x \in [0, 1]$ then (3.1.1) belongs to the same interval. We tested those models on our test set and we chose the one with not too high values for depth and width, but able to predict integrals with sufficient accuracy. In Tab. 3.1.1 we report the results we obtained. We can notice that the test error is often a bit smaller than the training one, this is probably due to the fact that the set of testing points is far smaller than the one used for training; indeed our training set is very large and it is able to cover many cases. Moreover, increasing the complexity of the model does not automatically lead to better results in terms of accuracy; usually that is because if the model is complex, it is difficult to train. However this is not our case, because all the models we created reached convergence in 3000 epochs, but as we will see later, there are oscillations that may influence the final result.

Besides we want to avoid too complex models because of the computational time for prediction. Indeed it is linearly proportional to the depth but quadratically to the width, so if the number of units for each hidden layer is very large, then our computational time will explode. In this case a good compromise between complexity and accuracy is given by the model composed by 3 layers and 100 units for each layer. After having chosen our hyperparameters, we have taken the 20% of the training dataset as validation to check there is no overfitting and we have retrained the model for 3000 iterations (epochs). As mentioned above, we can see in Fig. 3.1.1 that the MSE and MAE are oscillating; this is probably due to the choice of Adam as optimizer. Indeed it is really fast to approach a minimum, but then the exponential decay rates make the error oscillate. To be sure we have reached a (local) minimum, we have taken the weights coming from this model to initialize a new one with the same hyperparameters as before but that exploits SGD as optimizer and we have run it for 10000 epochs. As shown in Fig. 3.1.2, the error is not oscillating anymore and it has reached a stable state. Results are summarized in Tab. 3.1.2.

These results are not as good as we would have liked, in particular because this is a very simple case. Indeed, this is to be expected because, when we perform regression, it is really hard to obtain a mean squared error which is below $1\text{e}-09$. As a matter of fact, due to the strong non convexity of the minimization problem, we often attain local minima. Nonetheless, it could be relevant to mention that we actually tried to create a model to integrate

$$\int_{x_0}^{x_1} 1 \, dx. \tag{3.1.3}$$

The related MSE was $1\text{e}-16$, which corresponds to a mean absolute error of $1\text{e}-08$, but we definitely cannot do better than this because we are minimizing the mse and we have already reached machine precision.

Moreover, as the accuracy is limited by the minimization process only, we can expect it will not deteriorate in more complex situations.

Depth	Width	MSE train	MSE test	MAE test
2	50	3.977e − 07	3.042e − 07	3.780e − 04
	100	2.705e − 07	1.455e − 06	1.080e − 03
	150	5.715e − 08	3.453e − 08	1.414e − 04
	200	4.691e − 08	5.048e − 08	1.712e − 04
	250	1.215e − 07	4.443e − 08	1.398e − 04
	300	9.643e − 08	2.708e − 08	1.208e − 04
3	50	2.763e − 07	7.457e − 08	2.304e − 04
	100	3.132e − 08	1.244e − 08	8.291e − 05
	150	3.648e − 08	2.668e − 08	1.290e − 04
	200	5.970e − 09	4.275e − 09	5.051e − 05
	250	9.685e − 07	6.379e − 08	2.133e − 04
	300	2.255e − 08	1.480e − 08	8.657e − 05
4	50	5.426e − 08	3.413e − 07	5.411e − 04
	100	1.609e − 08	1.117e − 08	7.988e − 05
	150	2.296e − 08	1.695e − 08	1.053e − 04
	200	8.221e − 08	1.703e − 08	1.036e − 04
	250	1.173e − 06	5.117e − 08	1.782e − 04
	300	2.367e − 06	7.135e − 07	8.061e − 05
5	50	3.599e − 06	3.435e − 07	4.658e − 04
	100	8.574e − 08	8.466e − 08	2.474e − 04
	150	1.211e − 07	3.546e − 08	1.636e − 04
	200	5.132e − 08	4.470e − 08	1.861e − 04
	250	1.785e − 08	1.467e − 08	9.312e − 05
	300	1.004e − 06	1.311e − 06	1.043e − 03
6	50	4.533e − 08	1.056e − 08	7.507e − 05
	100	7.678e − 08	2.091e − 08	4.226e − 04
	150	1.500e − 08	1.176e − 08	7.051e − 05
	200	5.353e − 08	3.280e − 08	1.466e − 04
	250	5.159e − 08	1.830e − 08	1.125e − 04
	300	7.288e − 07	2.531e − 078	3.948e − 04

Table 3.1.1. Results in terms of mean squared error (MSE) and mean absolute error (MAE) for training and test with different numbers of layers (depth) and units per layers (width) to create a model to compute $\int_{x_0}^{x_1} x^p dx$.

	MSE train	MSE test	MAE train	MAE test
Adam	3.468e − 07	3.549e − 08	3.972e − 04	1.647e − 04
SGD	5.900e − 09	6.078e − 09	5.576e − 05	5.561e − 05

Table 3.1.2. Results in terms of mean square error (MSE) and mean absolute error (MAE) for training and test of a model to compute $\int_{x_0}^{x_1} x^p dx$ with depth= 3 and width= 100. The first one uses Adam as optimizer. The second one has been initialized with the weights coming from the first model and uses Stochastic Gradient Descend (SGD) as optimizer.

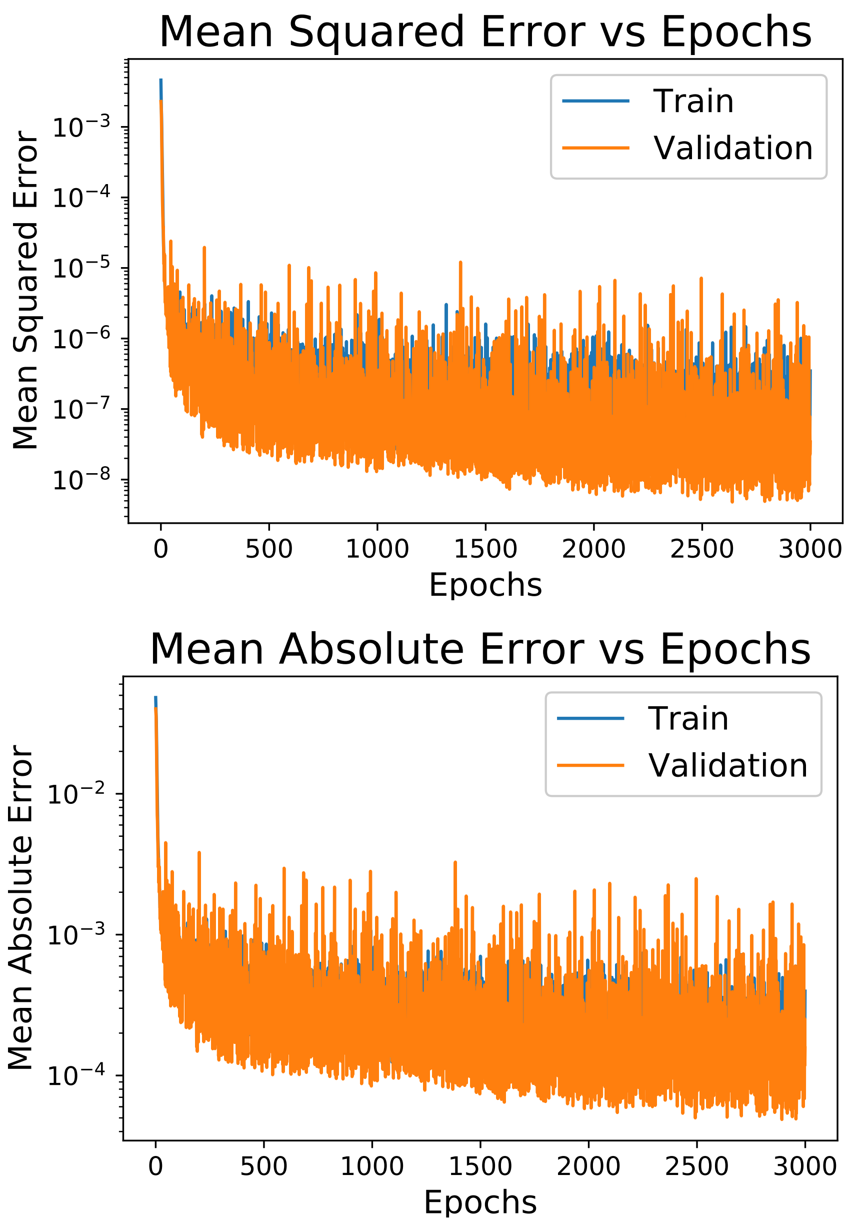


Figure 3.1.1. Semilogarithmic plot of (top) mean squared error (MSE) and (bottom) mean absolute error (MAE) of the training and the validation set of a model to compute $\int_{x_0}^{x_1} x^p dx$ with depth= 3, width= 100 and Adam optimizer. The training run for 3000 iterations (epochs)

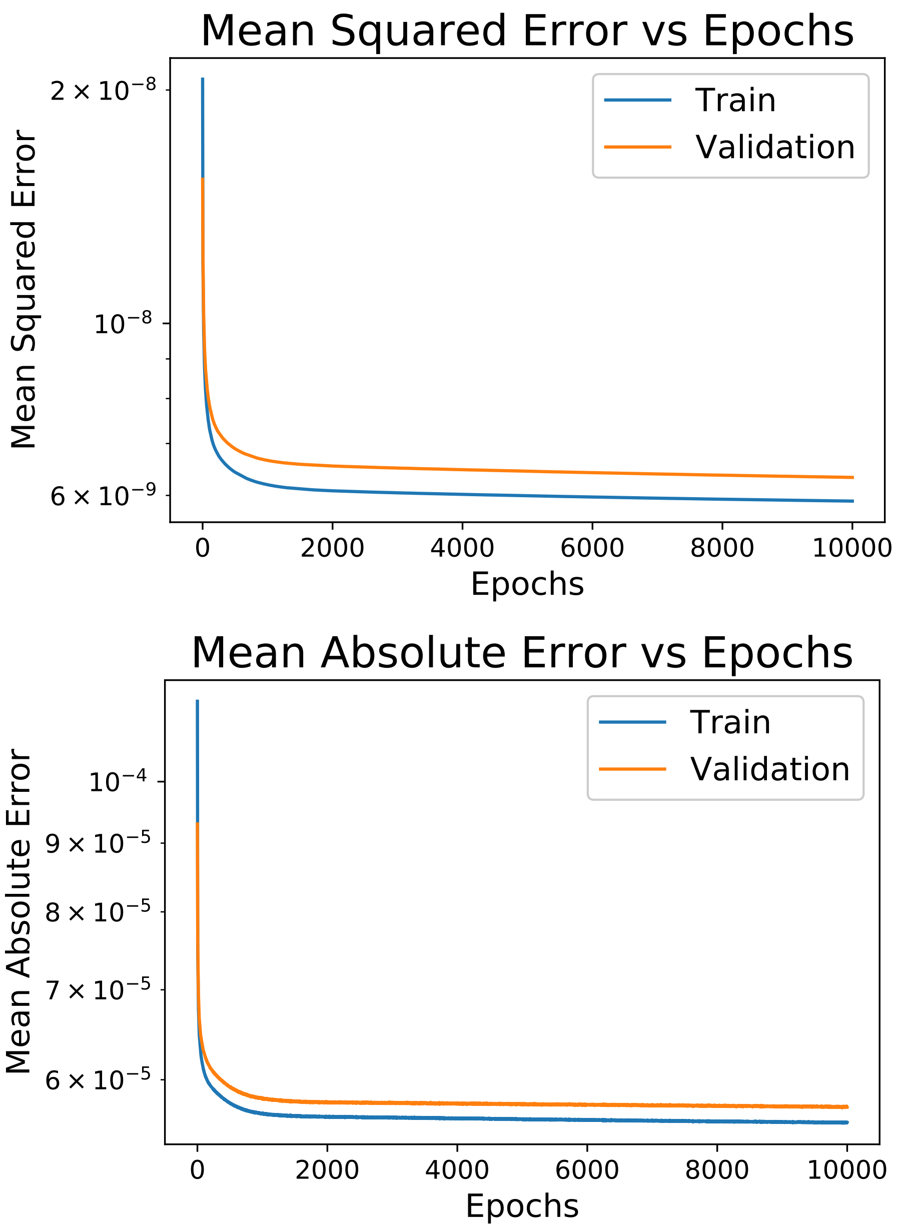


Figure 3.1.2. Semilogarithmic plot of (top) mean squared error (MSE) and (bottom) mean absolute error (MAE) of the training and the validation set of a model to compute $\int_{x_0}^{x_1} x^p dx$ with depth= 3, width= 100 and Stochastic Gradient Descend (SGD) optimizer and weights initialized using the previous model with Adam optimizer. The training run for 10000 iterations (epochs)

3.2 Model to compute Mass and Stiffness Matrix Entries with Control Points

Let us consider the following Poisson equation in a 2D domain

$$\begin{cases} -\Delta u &= f \quad \text{in } \Omega \\ \nabla u \cdot \boldsymbol{n} &= 0 \quad \text{on } \partial\Omega, \end{cases} \tag{3.2.1}$$

where Ω is the square $(0,1) \times (0,1)$ that has been cut. Moreover, we require $\int_\Omega f = 0$ as a compatibility condition. Equation (3.2.1) can be rewritten in the weak form as

$$\text{Find } u \in V : \int_\Omega \nabla u \cdot \nabla v = \int_\Omega fv \quad \forall v \in V, \tag{3.2.2}$$

where $V = H_0^1(\Omega) \setminus \mathrm{I\!R}$. In order to solve the problem we construct a suitable mesh $\mathcal{T}_h(\Omega)$ of size h, characterized by a collection of elements $Q \in \mathcal{T}_h(\Omega)$. The numerical solution $u_h \in H_0^1(\Omega)$ is a Bézier function when we restrict it to a single element, i.e.

$$u_h(x,y) = \sum_{i=0}^{3} \sum_{j=0}^{3} c_{i,j} B_{\{i,j\},p}(x,y), \tag{3.2.3}$$

where $B_{\{i,j\},p}(x,y) = \binom{p}{i} x^i (1-x)^{p-i} \binom{p}{j} y^j (1-y)^{p-j}$, with $p = 3$ in our numerical examples. The same can be done with v_h and f_h. This means they only differ in the control points $c_{i,j}$ and that if we are able to create a model to predict the mass and the stiffness matrices, then we can solve 3.2.1. Thus the aim of this experiment is to compute the following integrals

$$\int_\Omega B_{\{i,j\},p}(x,y) B_{\{k,l\},p}(x,y)\, dx\, dy \quad i,j,k,l = 0, \dots, p \tag{3.2.4}$$

and

$$\int_\Omega \nabla B_{\{i,j\},p}(x,y) \cdot \nabla B_{\{k,l\},p}(x,y)\, dx\, dy \quad i,j,k,l = 0, \dots, p, \tag{3.2.5}$$

which are the components of the mass (M) and the stiffness (S) matrix, respectively. M and $S \in \mathrm{I\!R}^{16 \times 16}$, so, in principle, we should predict 256 values, but they are symmetric and this means we only need 136 values to create them. Actually, using 1.1.7 and 1.1.9, integral 3.2.5 can be rewritten as follow

$$
\begin{aligned}
&\int_\Omega \left(B'_{i,p}(x)B'_{k,p}(x) + B'_{j,p}(y)B'_{l,p}(y)\right)\,dx\,dy = \\
&\int_\Omega (p^2\left(B_{i-1,p-1}(x) - B_{i,p-1}(x)\right)\left(B_{k-1,p-1}(x) - B_{k,p-1}(x)\right) + \\
&p^2\left(B_{j-1,p-1}(y) - B_{j,p-1}(y)\right)\left(B_{l-1,p-1}(y) - B_{l,p-1}(y)\right))\,dx\,dy = \\
&\int_\Omega \left(p^2\left(\frac{p-i+1}{p}B_{i-1,p}(x) + \frac{i}{p}B_{i,p}(x) - \frac{p-i}{p}B_{i,p}(x) - \frac{i+1}{p}B_{i+1,p}(x)\right)\right. \\
&\left(\frac{p-k+1}{p}B_{k-1,p}(x) + \frac{k}{p}B_{k,p}(x) - \frac{p-k}{p}B_{k,p}(x) - \frac{k+1}{p}B_{k+1,p}(x)\right) + \\
&p^2\left(\frac{p-j+1}{p}B_{j-1,p}(y) + \frac{j}{p}B_{j,p}(y) - \frac{p-j}{p}B_{j,p}(y) - \frac{j+1}{p}B_{j+1,p}(y)\right) + \\
&\left.\left(\frac{p-l+1}{p}B_{l-1,p}(y) + \frac{l}{p}B_{l,p}(y) - \frac{p-l}{p}B_{l,p}(y) - \frac{l+1}{p}B_{l+1,p}(y)\right)\right)\,dx\,dy,
\end{aligned}
\tag{3.2.6}
$$

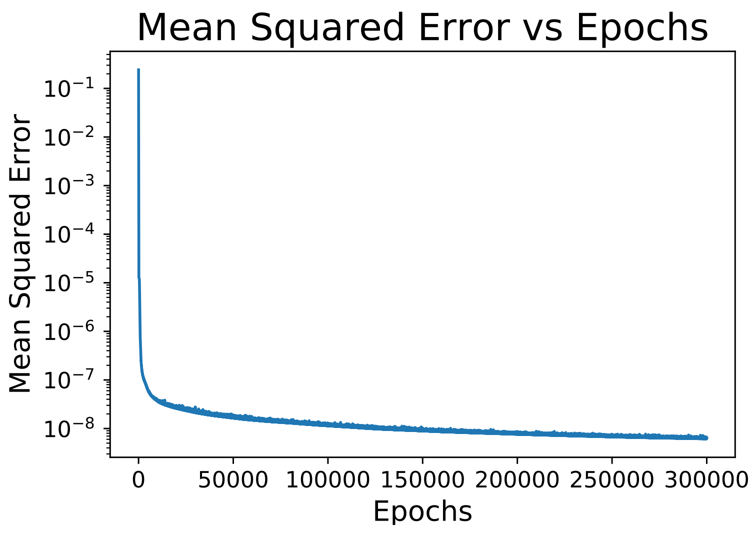
with $i, j, k, l = 0, \dots, p$. This means 3.2.5 can be computed with the same model we will construct for the mass matrix, because this is a sum of integrals of multiplied Bernstein polynomials of degree p.

Construction of the dataset based on control points Our very first attempt was to take 5690 Bézier functions of degree $p = 3$ that cut the domain from left to right as seen in Fig. 1.2.2 on the top left. The cut functions were of the form

$$C(x) = \sum_{i=0}^{3} c_i B_{i,p}(x), \tag{3.2.7}$$

and we made sure that each curve had no self-intersections and that was entirely inside the square $[0, 1] \times [0, 1]$. In order to create these curves we have considered all the possible combinations of 4 points of type: $\left\{\left(0, \frac{N_1}{10}\right), \left(\frac{1}{3}, \frac{N_2}{10}\right), \left(\frac{2}{3}, \frac{N_3}{10}\right), \left(1, \frac{N_2}{10}\right)\right\}$, with $N_1, N_2, N_3, N_4 = 0, \dots, 10$. Then we have constructed the Lagrangian curves of degree 3 able to interpolate our groups of 4 points and we have discarded the ones not entirely inside the square. In the end, exploiting a change of basis, we have converted the coefficients of the Lagrangian functions in such a way they were associated to Bézier ones.

Construction of the model for mass and stiffness matrix entries At this point, as training set we have used the four control points c_i characterizing each cutting curve and as labels the 136 exact integrals 3.2.4, computed as explained in Subsection 1.3.3, where Ω is the unitary square cut by the curves 3.2.7. We have created a model composed by 4 hidden layers with ReLU as activation function and 50 units each and an output layer with 136 units and Sigmoid as activation. Indeed Bézier functions in the square $(0, 1) \times (0, 1)$ have values between 0 and 1 and so when we multiply two of them the result still remains in that range. Then we have trained the model for 300000 iterations (epochs). In order to create the test set we have considered 900 groups of 4 points with coordinates $(0, y_1)$, $\left(\frac{1}{3}, y_2\right)$, $\left(\frac{2}{3}, y_3\right)$ and $(1, y_4)$, where y_1, y_2, y_3, y_4 were random numbers



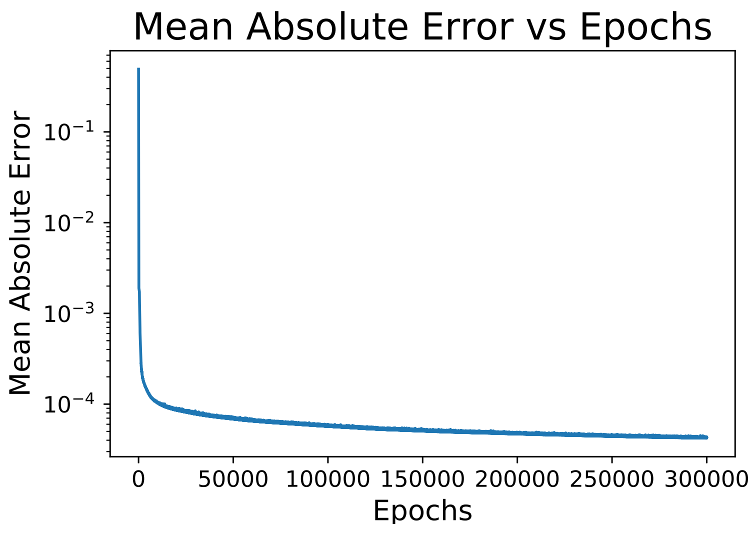


Figure 3.2.1. Semilogarithmic plot of (top) mean squared error (MSE) and (bottom) mean absolute error(MAE) of the training of a model to compute the mass matrix matrix of (3.2.2) with control points. We have set depth= 5 and width= 50 and 300000 iterations (epochs).

MSE train	MSE test	MAE train	MAE test
$6.270e-09$	$4.4408e-08$	$4.258e-05$	$5.673e-05$

Table 3.2.1. Results in terms of mean squared error (MSE) and mean absolute error (MAE) for training and test of a models to compute the mass matrix of (3.2.2) with control points. We have set depth= 5 and width= 50 and 300000 iterations (epochs).

$\in [0,1]$. As before we have constructed the Lagrangian curves that interpolated these points and discarded the ones not entirely in the unitary square or self-intersecting and performed a change of basis to obtain the Bézier coefficients. The behavior of the training error is in Fig. 3.2.1 and the results are in Tab. 3.2.1. We can notice that they are comparable to the ones obtained to integrate 3.1.1.

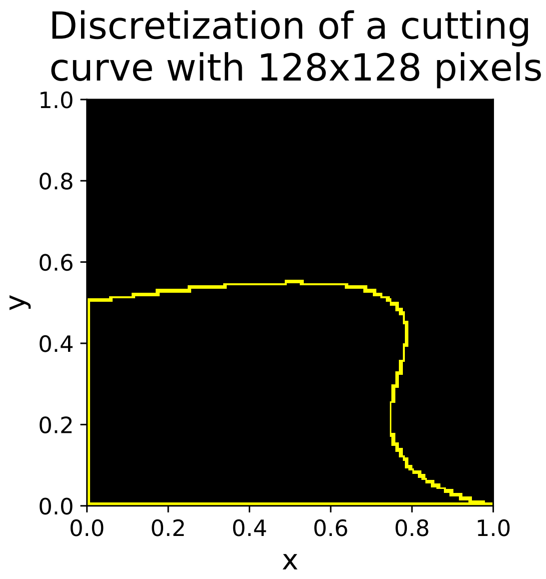


Figure 3.3.1. Image of 128×128 pixels that shows the cutting curve in yellow, together with the visible boundary of the element.

3.3 Model to compute Mass and Stiffness Matrix Entries with Images

Being encouraged by the previous experiment, we have tried to generalize it. First of all, we have considered the cutting curves of the the form

$$\boldsymbol{C}(x, y) = \sum_{i=0}^{3} \sum_{j=0}^{3} \boldsymbol{c}_{i,j} B_{i,j}(x, y), \tag{3.3.1}$$

which took into account all the situations in Fig. 1.2.2. In particular, for the cases on the top row in that figure, we have created Lagrangian curves interpolating points such that:

- the first interpolation point was $\left(0, \frac{N}{5}\right)$ where $N = 0, \ldots, 5$;
- the second and the third interpolation points were $\left(\frac{N_1}{5}, \frac{N_2}{5}\right)$ where $N_1, N_2 = 0, \ldots, 5$;
- the last interpolation point was $\left(1, \frac{N}{5}\right)$ where $N = 0, \ldots, 5$.

We took all the possible combinations of these four points and discarded the curves not entirely in $[0, 1] \times [0, 1]$ and the self-intersecting ones. We did the same procedure for the other two cases in Fig. 1.2.2. For the one on bottom left the last interpolation point was $\left(\frac{N}{5}, 0\right)$ where $N = 0, \ldots, 5$, for the case on bottom right the first and the last points had coordinates $\left(\frac{N}{5}, 0\right)$, with $N = 0, \ldots, 5$. Then we have divided our square $[0, 1] \times [0, 1]$ in 128×128 pixels and identified each one with a boolean value. In particular we have set $+1$ on cut elements and on the pixels on the visible boundary, in order to show which part of the domain is the active one, as shown in Fig. 3.3.1. Moreover we have rotated

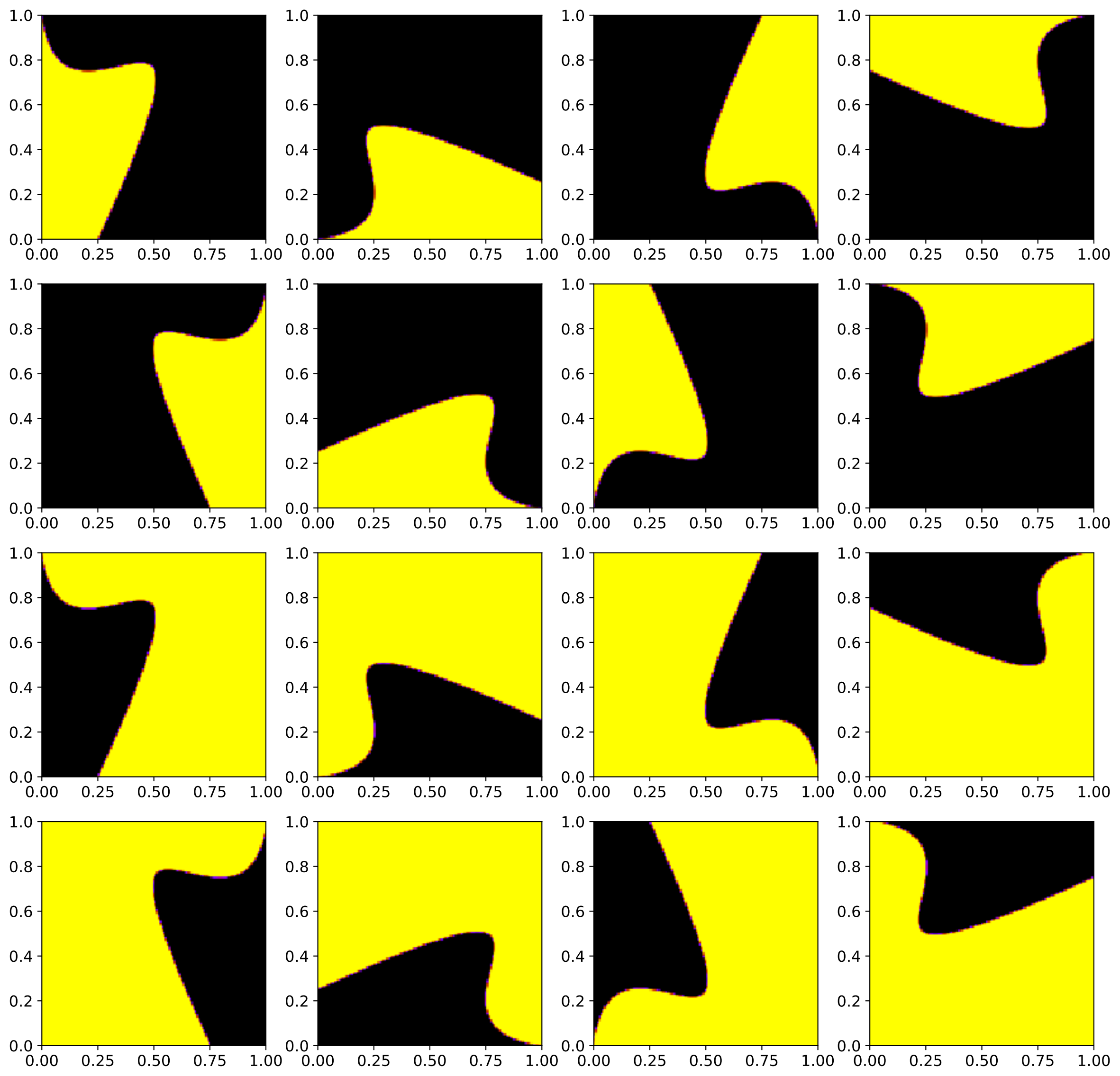


Figure 3.3.2. Given the image on the top left we rotate it by 90, 180 and 270 degrees, we take the symmetric with respect to $x = 0.5$ and we consider both the area under the curve and the one above. The yellow area is the visible one.

each cutting curve by 90, 180 and 270 degrees, considering both the area under the curve and the one above it and the symmetries with respect to the axes $x = 0.5$ and $y = 0.5$. In total, from each curve, we ended up with 16 images, as in Fig. 3.3.2. With this rough approximation some of the curves we had created gave us the same image, so we have reduced our dataset, considering only unique ones. In the end, our training set was composed by 12633 images of 128×128 pixels as input and as labels the exact integrals for the mass matrix, computed as in the previous example.

We have considered the ReLU activation function in the hidden layers and Sigmoid in the the output layer. Due to the fact we are dealing with images, we have decided to do convolution on top of our model in order to reach convergence faster, as explained in Chapter 2 Section 2.3. So we have a first convolutional layer with 10 channels and 3×3

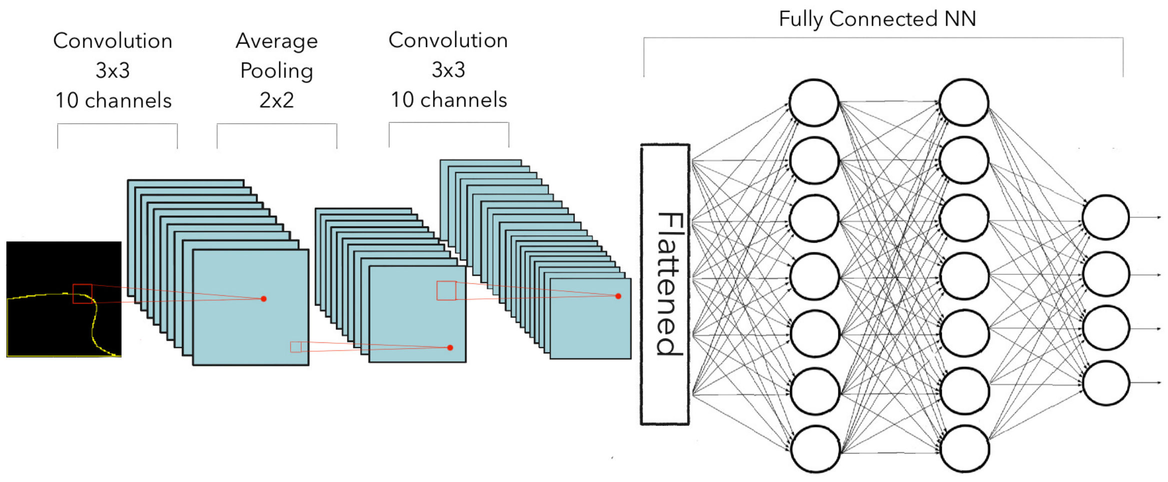


Figure 3.3.3. Model to compute the mass matrix with 2 convolutional layers, 1 average pooling, 2 hidden layers with width= 50. The training set is composed by images 128×128 showing the visible boundary.

MSE train	MSE test	MAE train	MAE test
8.174e − 08	1.3898e − 06	1.839e − 04	5.868e − 04

Table 3.3.1. Results in terms of mean squared error and mean absolute error for training and test of a model to compute the mass matrix of (3.2.2) with images. The training set is composed by images 128×128 showing the visible boundary. It has has 2 convolutional layers, 1 average pooling, 2 hidden layers with width= 50, run for 200 iterations (epochs).

mask, then an average pooling, another convolutional layer as before and 2 hidden layers with 50 units each. The output layer has 136 units and Sigmoid as activation function and runs for 200 iterations (epochs). The structure of this model is shown in Fig 3.3.3. The behavior of the training error is in Fig. 3.3.4 and the results are in Tab. 3.3.1.

3.4 The use of NN to compute matrix entries in an IGA code

The idea to use images to indicate visible areas is definitely useful because it takes a lot of effort to model cutting curves as Bézier functions, as we have seen in previous chapters. Using this approximation with pixels we avoid this reconstruction and we just need to sample our domain, which is very standard. The approximation we have done is quite rough and if we indicate with h the size of the pixel, i.e. $h = 1/128$, we cannot expect, in terms of mean absolute error, anything better than h^{-2}. So our method is performing rather well, in particular if we consider that this method can be easily expanded to 3D, where computing integrals on cut domains is still expensive and requires a lot of geometrical manipulations. Of course the idea to predict 136 values is quite demanding and the integrals we are computing are not taking into account the

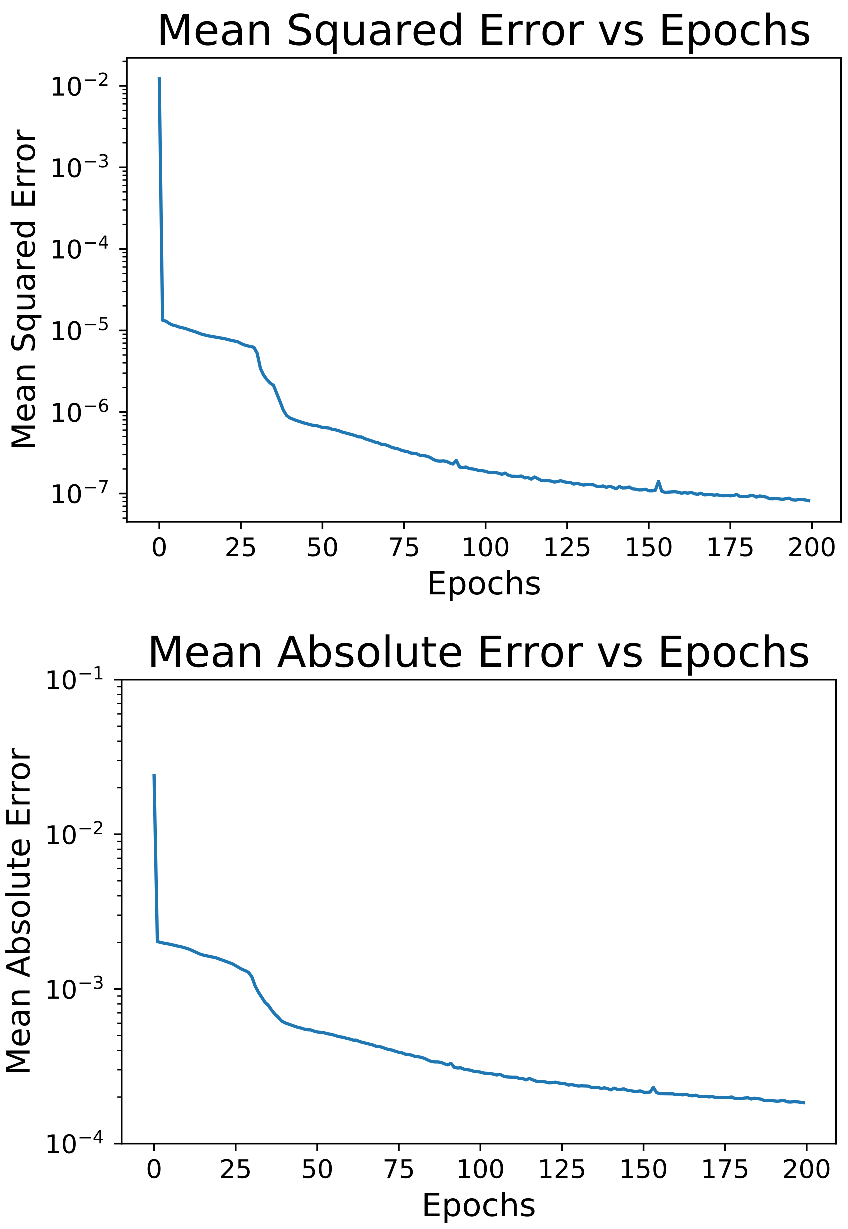


Figure 3.3.4. Semilogarithmic plot of (top) mean squared error (MSE) and (bottom) mean absolute error (MAE) of the training of a model to compute the mass matrix matrix of (3.2.2) with images. The training set is composed by images 128×128 showing the visible boundary. The model has 2 convolutional layers, 1 average pooling, 2 hidden layers with width= 50, run for 200 iterations (epochs).

presence of the determinant of a possible transformation of the domain. To be precise, let us suppose we want to solve (3.2.1) in a generic cut domain Ω. Then given a proper mesh $\mathcal{T}_h$, which is composed both by a collection of trimmed elements $Q^{\text{trim}} \in \mathcal{T}_h^{\text{trim}}$ and by non trimmed ones $Q^{\text{untr}} \in \mathcal{T}_h^{\text{untr}}$, the weak form for our numerical solution presents a bilinear form as follow

$$\begin{gathered} a(u_h, v_h) = \sum_{Q \in \mathcal{T}_h} a_Q(u_h, v_h) = \sum_{Q \in \mathcal{T}_h} \int_Q \nabla u_h \cdot \nabla v_h = \\ \sum_{Q^{\text{trim}} \in \mathcal{T}_h^{\text{trim}}} \int_{Q^{\text{trim}}} \nabla u_h \cdot \nabla v_h + \sum_{Q^{\text{untr}} \in \mathcal{T}_h^{\text{untr}}} \int_{Q^{\text{untr}}} \nabla u_h \cdot \nabla v_h \end{gathered} \tag{3.4.1}$$

where $a_Q(u_h, v_h)$ is the bilinear form restricted to an element $Q \in \mathcal{T}_h$ u_h and $v_h \in H_0^1$. Moreover, u_h and v_h restricted to the single element are Bézier curves. Recalling (3.2.6), it is enough to have a model to predict the mass matrix. Nonetheless, with the network we compute its entries in the parametric domain $\hat{\Omega}$ with mesh $\hat{\mathcal{T}}_h$, composed by elements $\hat{Q}$, i.e.

$$a_{\hat{Q}}(\hat{B}_{\{i,j\},p}, \hat{B}_{\{k,l\},p}) = \int_{\hat{Q}} \hat{B}_{\{i,j\},p}(\hat{\boldsymbol{x}}) \cdot \hat{B}_{\{i,j\},p}(\hat{\boldsymbol{x}})\, d\hat{\boldsymbol{x}}, \tag{3.4.2}$$

where $\hat{B}_{\{i,j\},p}$ denotes a basis function on the parametric element, but what we really need is

$$\begin{aligned} a_Q(B_{\{i,j\},p}, B_{\{k,l\},p}) &= \int_Q B_{\{i,j\},p}(\boldsymbol{x}) B_{\{i,j\},p}(\boldsymbol{x})\, d\boldsymbol{x} \\ &= \int_{\hat{Q}} B_{\{i,j\},p}(\boldsymbol{F}\hat{\boldsymbol{x}}) B_{\{i,j\},p}(\boldsymbol{F}\hat{\boldsymbol{x}})(\det \boldsymbol{F})(\hat{\boldsymbol{x}})\, d\hat{\boldsymbol{x}} \\ &= \int_{\hat{Q}} (B_{\{i,j\},p} \circ \boldsymbol{F})(\hat{\boldsymbol{x}})(B_{\{i,j\},p} \circ \boldsymbol{F})(\hat{\boldsymbol{x}})(\det \boldsymbol{F})(\hat{\boldsymbol{x}})\, d\hat{\boldsymbol{x}} \\ &= \int_{\hat{Q}} \hat{B}_{\{i,j\},p}(\hat{\boldsymbol{x}}) \hat{B}_{\{i,j\},p}(\hat{\boldsymbol{x}})(\det \boldsymbol{F})(\hat{\boldsymbol{x}})\, d\hat{\boldsymbol{x}} \end{aligned} \tag{3.4.3}$$

where $\boldsymbol{F}$ is the transformation from the physical to the parametric domain. This means that, to be able to use our network, we need the determinant of the transformation to be either constant element by element or slowly varying in order to have a good approximation if we compute it in the barycenter:

$$\begin{gathered} a_Q(B_{\{i,j\},p}, B_{\{k,l\},p}) \approx \tilde{a}_{\hat{Q}}(\hat{B}_{\{i,j\},p}, \hat{B}_{\{k,l\},p}) = \\ \int_{\hat{Q}} \hat{B}_{\{i,j\},p}(\hat{x}, \hat{y}) \cdot \hat{B}_{\{i,j\},p}(\hat{x}, \hat{y})(\det F)(\hat{x}_B, \hat{y}_B)\, d\hat{x} d\hat{y}, \end{gathered} \tag{3.4.4}$$

where $(\hat{x}_B, \hat{y}_B)$ are the coordinates of the barycenter.

Another idea is to consider the determinant as a Bézier curve. In this case we should create a model similar to the one of this experiment, by which we can predict the integral

$$\int_\Omega B_{\{i,j\},p}(x, y) B_{\{k,l\},p}(x, y) B_{\{m,n\},p}(x, y)\, dx\, dy \qquad i, j, k, l, m, n = 0, \dots, p. \tag{3.4.5}$$

Our methods work rather well for the prediction of the matrix entries, but they have some limitations. In particular, the assembly of the final matrix, that takes into account both trimmed and non trimmed elements, is very delicate and we must be careful in assigning the right value in the correct position. Moreover, when we run a simulation we must call the network every time we need to create the mass and stiffness matrix. Indeed, it is not believable to to save all the integrals before running the simulations because it would consume too much memory. So we need to take care of the computational time that is required to predict the results. Of course, images are slower to process than the control points of the curve but using convolutional networks with a small kernel as mask let us overcome this issue, as pointed out in Chapter 2 Section 2.3.

However, we could save even more time if we were able to compute the integrals off line. A successful idea, could be to try to predict quadrature points, instead of the actual values of the integrals; indeed it is definitely less expensive to save them and, once computed, in principle they can be used to solve any kind of problem. This is why in the next Chapter we will show the design of a NN to predict quadrature points.

Chapter 4

Design of NN to compute Quadrature Points

So far, the standard way to compute integrals is by using quadrature rules, as seen in Chapter 1.1, cf. Section 1.3. Thus a successful idea could be to predict quadrature points on trimming elements, instead of directly predicting the values of the integrals. In this way it is far much easier to make our model interact with existing codes to solve Partial Differential Equations (PDEs), like GeoPDEs [39]. Moreover, once we have placed the quadrature points, in principle we can solve any kind of problems and we can even map them for isogeometric ones. Moreover, it is possible to compute them off line because they do not arise any memory issue, which means that the computational time to evaluate the model does not play an important role anymore when we run a simulation. This is why we have focused our efforts in the creation of a model compute quadrature points on trimmed elements.

4.1 Model to compute Quadrature Points with Images

For this example, we have considered only cutting curves of types in Fig. 1.2.2. Indeed we have decided not to perform rotations because we wanted to impose in a strong way that when we rotate a curve we obtain the same rotated quadrature points. For this reason we have 2495 curves of which we have considered both the area under the curve and the one above it, ending up with 4990 images as Fig. 3.3.1. This choice implies that when we run a simulation we need to identify cutting elements, create the corresponding images with the visible boundary and then rotate them if they do not match with the cases considered in the training set. The output contains the quadrature points as x-coordinates, y-coordinates and the corresponding weights to compute 3.2.4. In particular, if we use polynomials of degree $p = 3$ as we did before, we need at least 4 quadrature points in each direction. Indeed Gauss-Legendre [40] with q quadrature points is able to exactly integrate polynomials of degree at most $2q - 1$. When we compute 3.2.4 if $B_i \in \mathbb{P}^p$ then $B_i B_k \in \mathbb{P}^{2p}$, which means

$$2q - 1 \geq 2p, \tag{4.1.1}$$

then

$$q \geq p + 1/2. \tag{4.1.2}$$

If $p = 3$ we need 4 quadrature points in each direction. In particular the 1/2 order of precision that remains is used for the determinant of the transformation that will be evaluated in the same quadrature points. The real issue with this model is that we don't know the exact quadrature points and weights, so we cannot minimize the mean squared error as we did in previous examples. We need to create a specific loss function which is able to tell us if the error we are committing in computing the integral with the quadrature rule is large or small. Thus we have assembled our custom loss function as follow

$$\frac{1}{(2p)^2}\sum_{i=0}^{2p}\sum_{j=0}^{2p}\frac{\frac{1}{m}\sum_{q=1}^{m}\left(\int_{\Omega_q} x^i y^j \, dx\, dy - \sum_{k=0}^{p} x_{k,q}^i y_{k,q}^j w_{k,q}\right)}{\left(\int_0^1\int_0^1 x^i y^j \, dx\, dy\right)^2}, \tag{4.1.3}$$

where p is the number of quadrature points and m is the length of the training set, i.e. the number of images with different cutting curves and visible areas. In this way we are trying to minimize the error we commit integrating monomials of order up to $2p$ using the integration points coming from the model. Indeed the goal is to compute 3.2.4 as best as we can with $B_i B_k \in \mathbb{P}^{2p}$. The term on the denominator is used to rescale the error; indeed, without it, when the area to integrate is very small, high order monomials become irrelevant in the computation of the error. In the end, as labels, we pass to the models the exact integrals of the monomials over the cut element, i.e.

$$\int_{\Omega_q} x^i y^j \, dx\, dy, \quad i, j = 0, \ldots, 2p. \tag{4.1.4}$$

4.1.1 NN for Quadrature Points in 1D

First of all we have tested our idea for the 1D case. This means that, taken the interval $[0, 1]$, we have cut it with 10000 random numbers $b \in (0, 1)$. We have computed the exact integrals as follow

$$\int_0^b x^i \, dx = \frac{b^{i+1}}{i+1}, \quad i = 0, \ldots, p, \tag{4.1.5}$$

with $p = 1$. The aim is to predict two quadrature points, so our output will have 4 units, 2 for the points and 2 for the corresponding weights. They belong to the interval $0, 1$ so we have selected Sigmoid as activation. The model had 6 hidden layers, with 30 units each and ReLU as activation function and it ran for 3000 iterations (epochs). In the end we can conclude the model performs very well, indeed let us recall that 1D Gauss quadrature points in the interval $[-1, 1]$ with $q = 2$ are $x_1 = -\frac{1}{\sqrt{3}}$ and $x_2 = \frac{1}{\sqrt{3}}$ and that the formula to determine them in a generic interval $[a, b]$ is

$$\tilde{x} = \frac{b-a}{2}x + \frac{b+a}{2}. \tag{4.1.6}$$

We have tested our model with different values of b and $a = 0$ and we have noticed that we recover Gauss quadrature points. The results are reported in Tab. 4.1.1.

b	x_1	x_2
0.5	0.1045	0.3936
0.75	0.1585	0.5917
1	0.2111	0.7890

Table 4.1.1. Quadrature points to integrate (4.1.5) coming from a model with 6 hidden layers, with 30 units each, run for 5000 iterations (epochs).

4.1.2 NN for Quadrature Points in Trimmed Domains

In order to compute (4.1.4), for each input image we have created a copy, which did not contain only binary numbers. Indeed we have assigned 0 to outside elements and +1 to inside ones, whereas to cut elements we have computed a greyscale, which indicated how large was the visible area for that element. Of course generating this kind of data has been demanding and we had computed this values of greyscale as explained in Subsection 1.3.3 for each element that contained the cutting curve. The advantage is that the computation of the loss function is more accurate but we actually pass to the model only the boundary of the visible area, so when we make predictions we do not need the greyscale images anymore. Basically (4.1.4) becomes

$$\int_{\Omega_q} x^i y^j \, dx \, dy = \sum_{s=0}^{n_p^2} x_s^i y_s^j g_s, \qquad i = 0, \dots, p, \tag{4.1.7}$$

where n_p is the number of pixels we have chosen, i.e. 128, x_s and y_s represent the coordinate of the center of the pixel and g_s the corresponding greyscale. On top of our model we have performed convolution with 10 channels and 3×3 mask, then an average pooling, another convolutional layer as before and 2 hidden layers with 50 units each and ReLU as activation. Unfortunately, even if we have created our own loss function, we still needed to set a metric, which will be irrelevant for our purposes. Indeed the output is composed by quadrature points and weights, whereas the labels are the correct integrals, so it does not make sense to compute the mean squared error. Still, we have set as metric mse and the number of units for the output layer must be equal to the length of the labels, which is the number of the monomials we are trying to integrate, i.e. $(2p + 1)^2 = 49$. Our real output has length $3(p + 1) = 48$, so the very last unit will just be ignored. Again we have chosen sigmoid as activation function for the last layer and we have run it for 400 iterations (epochs). The behavior of the loss function is in Fig. 4.1.1 and in Tab. 4.1.2 there are the results in terms of mean over all the test images of the absolute error for all the monomials we are checking, i.e.

$$\frac{1}{m} \sum_{q=1}^{m} \left(\int_{\Omega_q} x^i y^j \, dx \, dy - \sum_{k=0}^{p} x_{k,q}^i y_{k,q}^j w_{k,q} \right), \tag{4.1.8}$$

were m is the number of test images and $p = 3$, for $i, j = 1, \dots, 6$. The mean of the absolute error over all the monomials is 4.164e − 02.

If we show the quadrature points we obtain for an image taken from the test set, we can notice (Fig. 4.1.2) that they can lay outside the visible area. Indeed we have not

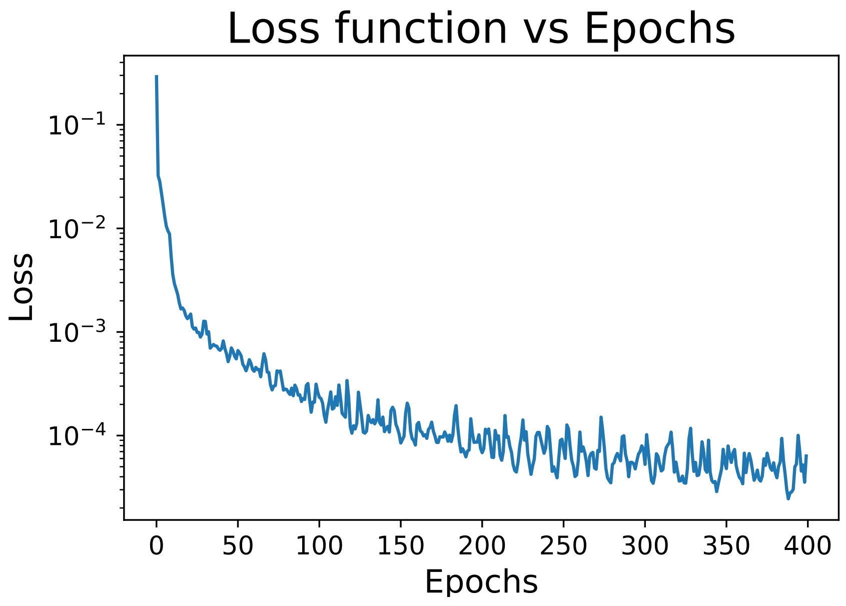


Figure 4.1.1. Semilogarithmic plot of the loss function (4.1.3) of a model to compute quadrature points. The training set is composed by images 128 × 128 showing the visible boundary. The model has 2 convolutional layers, 1 average pooling, 3 hidden layers with width= 50, run for 400 iterations (epochs).

$j\backslash i$	0	1	2	3	4	5	6
0	6.255e−02	1.864e−01	1.228e−01	1.229e−01	9.842e−02	8.196e−02	6.990e−02
1	2.649e−02	5.098e−02	5.073e−02	6.135e−02	4.914e−02	4.087e−02	3.474e−02
2	1.475e−02	1.198e−02	5.457e−02	4.079e−02	3.269e−02	2.725e−02	2.323e−02
3	9.659e−03	9.786e−03	4.091e−02	3.050e−02	2.441e−02	2.036e−02	1.738e−02
4	1.528e−01	9.486e−03	3.283e−02	2.445e−02	1.955e−02	1.631e−02	1.394e−02
5	4.885e−02	8.538e−02	2.742e−02	2.041e−02	1.633e−02	1.362e−02	1.165e−02
6	2.347e−02	2.999e−02	2.343e−02	1.745e−02	1.396e−02	1.165e−02	9.968e−03

Table 4.1.2. Mean over all the test images of the absolute error for all the monomials committed integrating with quadrature points and weights coming from a model with 2 convolutional layers, 1 average pooling, 3 hidden layers with width= 50, run for 400 iterations (epochs). The training set is composed by images 128 × 128 showing the visible boundary.

imposed them to be inside, the only requirement is that they integrate in a good way monomials in the square $[0, 1] \times [0, 1]$.

4.2 Intuitive Error Estimate

For what concerns the new quadrature formula obtained with the Neural Network, we do not have a clear error estimate, thus we cannot really produce a complete convergence theorem for our algorithm. Nonetheless, we can provide an intuition of what we expect to see in numerical results.

First of all, we must point out that the NN quadrature formula is used only on trimmed elements, whereas untrimmed ones are treated in a standard way. Given a

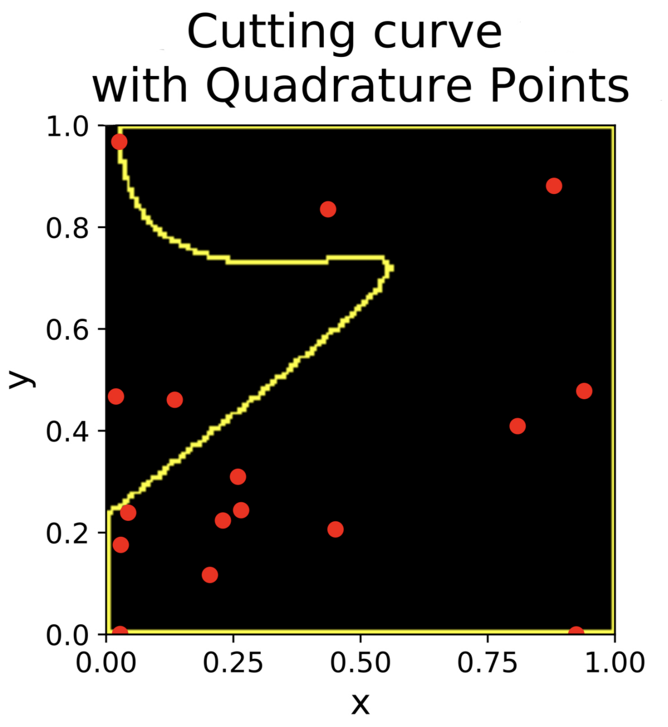


Figure 4.1.2. Cutting curve with quadrature points computed with a model with 2 convolutional layers, 1 average pooling, 3 hidden layers with width= 50, run for 400 epochs.

domain Ω with a mesh $\mathcal{T}_h$ where h is the mesh size, then it is natural to assume that the number of cut elements is proportional to $1/h$. The weak form of the problem we want to solve is

$$\text{Find } u \in V : a(u,v) = (f,v) \quad \forall v \in V = H_0^1(\Omega). \tag{4.2.1}$$

If we consider our numerical solution $u_h \in V_h \subseteq H_0^1(\Omega)$ where V_h is defined such that u_h is a Bézier curve when restricted to the single element $Q \in \mathcal{T}_h$, then the numerical problem becomes

$$\text{Find } u_h \in V_h : a_h(u_h, v_h) = (f, v_h)_h \quad \forall v_h \in V_h, \tag{4.2.2}$$

where $a_h(\cdot,\cdot)$ and $(f,v)_h$ are obtained by replacing the integrals with a quadrature formula. Let $a_{h,Q}(\cdot,\cdot)$ and $(f,v)_{h,Q}$ be the bilinear form and the right hand side restricted to the element Q. To fix ideas we consider

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v, \tag{4.2.3}$$

then, if Q is untrimmed, $a_{h,Q}(u_h, v_h) = \int_Q \nabla u_h \cdot \nabla v_h$, whereas if it is trimmed $a_{h,Q}(u_h, v_h) = \int_{h,Q} \nabla u_h \cdot \nabla v_h$ where $\int_{h,Q}$ denotes the quadrature rule on Q.

Recalling the first Strang Lemma [40], we have:

$$\begin{aligned}
\|u-u_h\| \leq & \inf_{w_h\in V_h}\left[\|y-w_h\| + \sup_{v_h\in V_h, v_h\neq 0}\frac{a(w_h,v_h)-a_h(w_h,v_h)}{\|v_h\|}\right] \\
& + \sup_{v_h\in V_h, v_h\neq 0}\frac{|(f,v_h)-(f,v_h)_h|}{\|v_h\|} \\
\leq & \inf_{w_h\in V_h}\left[\|y-w_h\| + \sum_{Q\in\mathcal{T}_h^{\text{trim}}}\frac{(a_Q-a_{h,Q})(w_h,v_h)}{\|v_h\|}\right] \\
& + \sum_{Q\in\mathcal{T}_h^{\text{trim}}}\frac{|\int_Q fv_h - \int_{h,Q} fv_h|}{\|v_h\|}.
\end{aligned} \tag{4.2.4}$$

If we indicate with E_{NN} the maximum integration error done by our NN, then, due to scaling, we expect all terms within the summations to be bounded by $h^2\mathrm{E}_{\mathrm{NN}}$. As we sum over $1/h$ elements, the final estimate we may expect is:

$$\|u-u_h\| \leq c_1h^p + c_2h\mathrm{E}_{\mathrm{NN}}, \tag{4.2.5}$$

where p is the degree of the basis functions and c_1 and c_2 are positive constants that do not depend on h.

4.3 Conclusions

The proposed method is very promising because it is easily embeddable in existing codes, like GeoPDEs [39], and it gives us the possibility to solve almost every problem. Indeed, quadrature points do not change if we consider a different function to integrate, so we can directly compute the mass and the stiffness entries without rearranging terms, which could be a laborious and very delicate process. The phase of preprocessing is quite simple, indeed creating images of 0 and 1 given a domain is straightforward both in 2D and in 3D, it only remains to rotate images that does not match with the cases used for the training. This is not complicated and it assures we obtain a consistent method, because we need to obtain the exact rotated points if we rotate an image.

However, the fact that those points do not lay completely inside the visible area is really not an ideal case; one way to solve this issue could be to add to the loss function a penalization if the model finds points outside, but our first attempts in this sense show that the convergence of our new loss is very poor. We believe that the reason for this is quite simple: we initialize our network with random weights, thus at the beginning we are far from the global minimum and we likely remain stuck in a local one. Moreover the optimization problem is highly non convex. Otherwise a strategy to strongly force the points to stay inside should be found.

Nonetheless our basis functions exist on the whole square, so it is not a big issue to compute them outside the visible area. The right hand side might not be defined, even if in many problems the data are simple quantities, i.e. gravity, or can be easily extended.

Conclusions

In this works we have proposed innovative techniques for the integration over trimming surfaces that result in the numerical approximation of partial differential equations.

In order to represent arbitrary surfaces, a regular surface is cut by a cutting curve, but we do not know the exact shape of the trimming function, which means that it has to be approximated. Usually, Bézier basis functions are employed to represent the curve, but this implies a great effort in terms of geometrical computations as seen in 1 Section 1.2, in particular in a 3D domain. Indeed we need to create very fine meshes in order to find interpolations points to perform the reconstruction [34].

The idea is to limit the number of these geometrical operations, avoiding the reconstruction phase and using techniques coming from the machine learning world. This is a very fast growing field and as we have seen in Chapter 2 it offers many possibilities even in numerical analysis. So far it has almost never been used due to the fact that it is still not as precise as other methods, i.e. Galerkin, and a solid theoretical background has not been developed yet. Nonetheless, the advantages it could offer in terms of computational time, especially in 3D, could make it become an extremely powerful instrument to be exploited. Usually, these models are very difficult to train, first of all because of choice of the dataset that must be very accurate and whose creation could be expensive; moreover the optimization in the training phase could require many iterations or epochs to reach a stable state.

Neural Networks are able to learn internal rules given a certain input and a teacher signal and they have the property to be universal approximators for functions. Of course, being an almost unexplored field we have started our analysis with a very simple 1D example, like the integration of x^p over $[x_0, x_1] \subset [0, 1]$. As we have seen in Chapter 3, the results are not as good as we would have expected, given the simplicity of the function to integrate. In particular, the main problem of machine learning seems to be the fact that it attains a local minimum and it is not able to overcome this issue. Indeed, it can be noticed that it is difficult to gain a MSE smaller than $1\mathrm{e}-9$.

We have not let these results discourage us and we have tried to create models to predict the mass and the stiffness matrix for a PDE, see Chapter 3. The results have been promising because the corresponding MSE is similar to the one obtained for the simpler case. We have used two different approaches to accomplish this task. The first still needs a re-parameterization of the cutting curve. Indeed we have created several curves able to span our parametric domain $[0, 1] \times [0, 1]$ and we have passed as input to the model the control points of these curves. Of course this could be interesting but does not solve the problem we need to compute laborious geometrical operations to have the reconstruction of the trimming. So we have completely changed perspective and we have passed images to the model as input, showing the cutting curve and the visible boundary. Creating these kind of images is straightforward both in 2D and in 3D; indeed it is enough to divide the domain in pixels and assign value +1 to elements containing the cutting curve or on the boundary. This could look like a very rough approximation, in particular if it is considered that our element has 128×128 pixels, but it actually gives us good results and, thanks to the use of Convolutional Networks, it does not require a lot of time to train.

Of course, these models have some limitations. First of all we directly predict the values of the integral, without taking into account the presence of a possible transformation from the parametric to the physical domain. This problem could be overcome in case of identity or constant determinant of the transformation, but otherwise we must be conscious we are introducing further approximations. Moreover the predictions have to be done runtime due to memory issues. It is not possible to save all the matrices for every single cut element, so the computational time to run the model must be taken into account. Finally the assembly of the matrices that contain both trimmed and untrimmed elements is a very delicate operation and it is not so easy to make our model interact with existing codes that rely upon quadrature points.

Thus a new model for the prediction of quadrature points has been created in Chapter 4. Not only it is far much easier to embed in existing codes like GeoPDEs, but the points can be computed offline so that the computational time is not relevant anymore. Moreover, in principle those points can be used to solve any kind of problem so they open infinite opportunities. Again the network has been created using images, that have proved to be a suitable tool in previous experiments. We had to construct a loss function that made us minimize the error committed in integrating monomials up to order 6 in our domain $[0,1] \times [0,1]$. Actually the results are not entirely satisfactory. First of all because we obtain points are not entirely contained in the active area. This is not ideal even if the basis functions exist all over the domain and in many problems the right hand side can be easily extended, but in future works it is better to try to impose a penalization in the loss when points lay outside or to find a way to strongly force them to stay inside. The first attempts in this sense has not given good results, probably because the problem is highly non convex and the optimization is very hard. Moreover, for instance, when we are computing the entries for the stiffness matrix we are not integrating monomials, but polynomials. This means that our final error estimate could be multiplied by a constant that makes it worse than the one obtained for monomials.

Still, we hope that this work opens the possibility to improve these models, in particular introducing the 3D case. Indeed, they have been created with the aim to extend them in three dimensions in future. The idea to use images should simplify the final prediction, but the creation of a suitable dataset is still an open problem. First of all because it is not possible to divide the cube $[0,1] \times [0,1] \times [0,1]$ uniformly taking all the possible cases because the dataset would be too big. Indeed there would be memory issues and it would be extremely difficult to train. So a good strategy to create the most likely curves that can cut the domain has to be found. Moreover we have trained our model to find quadrature points using the exact integrals in the loss function, so it would be very laborious to compute them in 3D, but it must be taken into account that if the network works well this is done once and for all.

Ringraziamenti

Per prima cosa voglio ringraziare mia madre per avere sempre creduto in me, per essere stata la mia più grande sostenitrice e la mia roccia in tutto questo percorso e per avermi permesso di realizzare tutti i miei sogni. Ringrazio mio padre che mi ha trasmesso tutta la sua grinta e la sua determinazione e che sono certa sarebbe orgoglioso di potermi vedere oggi. Mia sorella che mi ha sempre spronato e che è come una seconda madre per me.

Un grazie speciale a Franco per avermi sostenuto, anche nei momenti di sconforto. Grazie ad Andrea A. e al mio piccolo Alberto per avermi sempre fatto sorridere e svagare quando ne avevo bisogno, ai miei nonni per avermi sempre accolto. Grazie ad Andrea R. per esserci sempre!

Vorrei inoltre ringraziare le mie relatrici: la Prof. Annalisa Buffa e la Prof. Paola Francesca Antonietti per avermi permesso di lavorare su un argomento innovativo e stimolante e per essere state incoraggianti e disponibili. Infine, non posso non ringraziare infinitamente i collaboratori della Prof. Buffa, Pablo e Ondine, per essere stati estremamente presenti e per avere avuto idee meravigliose, che hanno reso questa tesi più ricca e interessante.

Bibliography

[1] T.J.R. Hughes, J.A. Cottrell, Y.Bazilevs (2005). *Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement.* Comput. Methods Appl. Mech. Engrg., 194:4135-4195.

[2] L. Beirão da Veiga, A. Buffa, G. Sangalli, R. Vázquez (2014). *Mathematical analysis of variational isogeometic methods.* Acta Numer., 23:157-287.

[3] I.C.Braid (1974). *Designing with Volumes.* Cantab. Press., Cambridge.

[4] B. Marussig, T.J.R. Hughes (2017). *A Review of Trimming in Isogeometric Analysis: Challenges, Data Exchange and Simulation Aspects.* The Institute for Computational Engineering and Sciences, The University of Texas at Austin.

[5] C. de Boor (2001). *A practical guide to splines*, Applied Mathematical Sciences, vol. 27. Springer, New Yotk.

[6] G. Farin (2002). *Curves and surfaces for CAGD: a practical guide.* Morgan Kaufmann, San Francisco, Calif.[u.a.].

[7] F. Auricchio, L. Berãio Da Veiga, T.J.R. Hughes, A. Reali, G. Sangalli (2010). *Isogeometric collocation methods.* Mathematical Models and Methods in Applied Sciences 20(11), 2075-2107.

[8] K. Li, X. Qian (2011). *Isogeometric analysis and shape optimization via boundary integral.* Computer-Aided Design 43(11), 1427-1437.

[9] T.W. Sederberg, T. Nishita (1990). *Curve intersection using Bézier clipping.* Computer-Aided Design 22(9), 534-549.

[10] H.J. Kim, Y.D. Seo, S.K. Youn (1999). *Isogeometric analysis for trimmed CAD surfaces.* Computer Methods in Applied Mechanics and Engineering 198(37-40), 2982-2995.

[11] H.J. Kim, Y.D. Seo, S.K. Youn (2010). *Isogeometric analysis with trimming technique for problems of arbitrary complex topology.* Computer Methods in Applied Mechanics and Engineering 199(45-48), 2796-2812.

[12] B. Marussig (2016). *Seamless Integration of Design and Analysis through Boundary Integral Equations.* Monographic Series TU Graz: Structural Analysis. Verlag der Technischen Universität Graz.

[13] B. Marussig, J. Zechner, G. Beer, T.P. Fries (2016). *Stable isogeometric analysis of trimmed geometries.* Computer Methods in Applied Mechanics and Engineering.

[14] R. Schmidt, R. Wüchner, K.U. Bletzinger (2012). *Isogeometric analysis of trimmed NURBS geometries.* Computer Methods in Applied Mechanics and Engineering 241-244, 93-111.

[15] A.P. Nagy, D.J. Benson (2015). *On the numerical integration of trimmed isogeometric elements.* Computer Methods in Applied Mechanics and Engineering 284, 165-185.

[16] Y. Wang, D.J. Benson (2016). *Geometrically constrained isogeometric parameterized level-set based topology optimization via trimmed elements.* Frontiers of Mechanical Engineering pp. 1-16.

[17] Y. Wang, D.J. Benson, A.P. Nagy (2015). *A multi-patch non-singular isogeometric boundary element method using trimmed elements.* Computational Mechanics 56(1), 173-191.

[18] J. Lasserre (1998). *Integration on a convec polytope.* Proceedings of the American Mathematical Society 126(8), 2433-2441.

[19] Y. Guo, M. Ruess, D. Schillinger (2016). *A parameter-free variational coupling approach for trimmed isogeometric thin shells.* Computational Mechanics 56(1), 173-191.

[20] E. Rank, M. Ruess, S. Kollmannsberger, D. Schillinger, A. Düster (2012). *Geometric modeling, isogeometric analysis and the finite cell method.* Computer Methods in Applied Mechanics and Engineering 249-252, 104-105.

[21] M. Ruess, D. Schillinger, Y. Bazilevs, V. Varduhn, E. Rank (2013). *Weakly enforced essential boundary conditions for NURBS-embedded and trimmed NURBS geometries on the basis of finite cell method.* International Jornal for Numerical Methods in Engineering 95(10), 811-846.

[22] M. Ruess, D. Schillinger, A.I. Özcan, E. Rank (2014). *Weak coupling for isogeometric analysis of non-matching and trimmed multi-patch geometries.* Computer Methods in Applied Mechanics and Engineering 269, 46-71.

[23] Y.D. Seo, H.J. Kim, S.K. Youn (2010). *Definition of "overfitting"* at `https://www.oxforddictionaries.com/`: this definition is specifically for statistics.

[24] I. Goodfellow, Y. Bengio, A. Courville (2016). *Deep Learning.* MIT Press, `http://www.deeplearningbook.org`.

[25] Y. LeCun, C. Cortes, C.J.C. Burges (2004). *THE MNIST DATABASE of handwritten digits.* `http://yann.lecun.com/exdb/mnist/`.

[26] K. Funahashi (1989). *On the approximate realization of continuous mappings by neural networks.* Neural Netw. 2, 183-192.

[27] K. Hornik, M. Stinchcombe, H. White (2016). *Multilayer feedforward networks are universal approximators*. Neural Netw. 2, 359-366.

[28] Y. LeCun, Y. Bengio, G.E. Hinton, R.J. Williams (2015). *Deep Learning*. Nature 521, 436-444.

[29] G. Elber, F. Massarwi (2016). *A B-spline based framework for volumerti object modeling*. Computer Aided Design, 78, 36-47.

[30] A. Buffa, R. Puppi, R. Vzquez (2019). *A minimal stabilization procedure for Isogeometric methods on trimmed geometries*. arXiv:1902.04937 [math].

[31] E. Burman, P. Hansbo (2012). *Fictitious domain finite element methods using cut elements: II. A stabilized Nitsche method*. Appl. Numer. Math., 62(4), 328-341.

[32] E. Burman, P. Hansbo (2014). *Fictitious domain methods using cut elements: III. A stabilized Nitsche method for Stokes' problem.*. ESAIM Math. Model. Numer. Anal., 48(3), 859-874.

[33] Open CASCADE SAS. (2018) *Open CASCADE 7.3.0*. `https://www.opencascade.com/`.

[34] P. Antolin, A. Buffa, M. Martinelli (2019). *Isogeometric Analysis on V-reps: first results*. arXiv:1903.03362v1 [math].

[35] R. Adams, J. Fournier (2003). *Sobolev Spaces, Volume 140*. Elsevier.

[36] C. Geuzaine, J.F. Remacle (2009). *Gmsh: a three dimensional finite element mesh generator with built-in pre- and post-processing facilities*. Internat. J. Numer. Methods Engrg., 79(11), 1309-1331.

[37] S.Boyd, L. Vandenberghe (2004). *Convex Optimization*. Cambrisge Univeristy Press, New York, `http://stanford.edu/~boyd/cvxbook/`.

[38] D.P. Kingma, J.L. Ba (2017). *Adam: A method for stochastic optimization*. Cornell University.

[39] R. Vazquez (2011). *GeoPDEs 3.1.0*. `http://rafavzqz.github.io/geopdes/`.

[40] A. Quarteroni, A. Valli (1994). *Numerical Approximation of Partial Differential Equations*. Springer.