An adaptive isogeometric finite element analysis

Kjetil André Johannessen

Master of Science in Physics and Mathematics

Submission date:	June 2009
Supervisor:	Trond Kvamsdal, MATH

Norwegian University of Science and Technology
Department of Mathematical Sciences

Problem Description

In this thesis we will explore the possibilities of making an adaptive solver for partial differential equations using finite element method with splines basis functions in an isogeometric environment. A fully automatic a posteriori error estimation will be utilized to identify large-error elements, followed by T-spline refinement of the mesh. The report shall include one or more illustrative examples.

Assignment given: 19. January 2009
Supervisor: Trond Kvamsdal, MATH

Abstract

In this thesis we will explore the possibilities of making a finite element solver for partial differential equations using the isogeometric framework established by Hughes et al. Whereas general B-splines and NURBS only allow for tensor product refinement, a new technology called T-splines will open for true local refinement. We will give an introduction into T-splines along with B-splines and NURBS on which they are built, presenting as well a refinement algorithm which will preserve the exact geometry of the T-spline and allow for more control points in the mesh. For the solver we will apply a residual-based a posteriori error estimator to identify elements which contribute the most to the error, which in turn allows for a fully automatic adaptive refinement scheme. The performance of the T-splines is shown to be superior on problems which contains singularities when compared with more traditional splines. Moreover the T-splines along with a posteriori error estimators are shown to have a very positive effect on badly parametrized models, as it seem to make the solution grid independent of the original parametrization.

Acknowledgments

It is a pleasure to acknowledge the generous help of a number of people in the preparation of this work. First of all, I am in debt to Trond Kvamsdal who has been my advisor throughout this thesis. Without his constant feedback, advice, inspiration and guidance, this thesis would never have seen the light of day. Also a large thanks to Thomas Hughes for some very inspiring response to my work and interesting conversations on his visit to NTNU in the spring of 2009. I would also take this opportunity to acknowledge Thomas Sederberg, Mike Scott, Nicholas North and all the other helpful people at the official T-spline web forum. You have been an unparalleled source of knowledge in anything T-spline related, and I have profited much from many interesting discussions on this forum. Many thanks to my fellow students Håkon Berg Johnsen, Siri-Malén Høynes and Magnus Dahler Norling who read and commented my entire thesis. Last, but not least, I would like to thank my family Linda and Victoria for their continued support and understanding during late working hours. Love does not make the world go 'round, it's what makes the ride worthwhile.

Contents

Chapter 1

Introduction

1.1 Computer-aided engineering

Computer-aided engineering (CAE) is being applied everywhere. From the dawn of computers, it has been used for mathematical computations and solving engineering problems. Over the years, not only has the hardware seen drastic improvements by the well-known Moore's Law, but the techniques and algorithms have seen vast improvements as well. One of these algorithms which was developed in the 1950's was the finite element method (FEM). It has been studied and refined ever since, resulting in countless books and articles on the method. Today, it is such a well-established method that it is found everywhere. FEM is taught at the university and applied in every aspect of the industry. Ranging from fluid flow calculations on nuclear submarines to medical simulations of the human circulatory system [22], the finite element method is applied to supply engineers, scientists and doctors with the most accurate descriptions of the forces and fluids in effect.

With the advancements in technology comes the desire to create more and more complex constructions. The vehicles and machines that is being created today are of such vast complexity that it is almost hopeless to keep track of everything, let alone do any numerical computations. For a comparison, a regular personal automobile has around 3 000 parts, while a Boeing 777 has around 100 000 parts [8]. If we look at nuclear submarines, they consists of over a million parts. Clearly these huge constructions require efficient handling at all stages of the design, analysis and construction. This is currently not the case as a severe bottleneck has appeared in later years which is requiring attention.

1.2 Computer-aided design

Computer-aided design (CAD) has its origins at a slightly later point in history than the CAE and FEM has. It is generally agreed [8] that it was first started by the two French automotive engineers Pierre Bézier of Renault and Paul de Faget de Casteljau of Citröen in 1966. The modern B-splines were first established in the PhD dissertation of Reisenfeld in 1972 [18] and it's rational counterpart the NURBS was not established before 1975 by Versprille [21]. One of the reasons for the CAE developing so much sooner than the CAD is that the concept of efficient design using a computer is heavily dependent on the existence of a graphical user interface (GUI). While pure number-crunching was present from the very cradle of the computer age, the graphical user interfaces was not to see widespread use before the 1970's. Once it arrived however, it quickly evolved and was accepted, or in many ways required, on any personal or industrial computer. By the time that the CAD community had started evolving, the analysis community had decades of experience and had no intention of changing their core systems and algorithms.

From then on the community for analysis and design evolved quite independent. This is largely due to the different requirements they each face. While the designers kept focus on systems which would be easy to manipulate, visualize and construct, the analyst kept focus on systems which were accurate, computationally fast and easy to interpret. The development has continued in both fields resulting in different systems relying on different geometric constructs. After quite some time with research in both these fields, the differences only seem to increase. This was fine since there have traditionally been very simple geometries which was feasible to do analysis on and thus the generation of the geometry itself could be done by hand. With increased computational power, came the desire to model more and more complex models which led to an unexpected halt. What was discovered was that it was no longer the running time of the analysis algorithms which posed the limit on what problems was possible to solve on a computer. It was the generation of the geometry mesh.

To perform any analysis, the model which is given from the designer would have to be converted to a suitable model ready for analysis. This process of conversion is far from trivial. The first step of conversion would be to create a model in which analysis is *possible*, while the second step would involve creating a model which is *good*. It is estimated [14] that as much as 20 percent of the total analysis time is spent on the first step, and as much as 60 percent of the time on the second step in the automotive, aerospace and ship building industries. This totals a 80/20 factor of conversion versus actual analysis. Needless to say, this imposes a severe bottleneck on the process and it is a strong wish from the community to decrease this factor.

1.3 State-of-the-art

There have been put a lot of research into the topic of automating the process of conversion from one system to the other. Both when it comes to creating a possible model by representing the geometry different, and creating a good model with appropriate refinements around singularities. Several algorithms and techniques have been developed, but none of these seem to have any widespread industrial success. This is largely due to the fact that the analysts have little confidence in these automatic mesh generators and still prefer to make grids by hand. That is not to say that automatic mesh generators are vacant in the industry. They are being implemented and used, but seldom alone. The mesh generators are often used as a preprocessing step before the analysts modify the mesh by hand to make it analysis suitable. When it comes to the second step of creating a *good* model, the problem gets harder. This is due to the fact that existing analysis models are only approximating the design model and refinement creates the need for communication with the design model. This direct communication is often impossible since the information regarding the exact geometry is lost in the original conversion. Hence this topic is even less subject to automated procedures than the original conversion.

1.4 Fundamental changes

With so much research going into the subject, and so few results, there were several people who started to suspect that the problem were lying somewhere else. While research previously had been dealing with solving the problem of automatic conversion and refining, the real cause of this question being so hard, lies somewhere else. Thomas Hughes at the University of Texas at Austin, had after discussing with a designer come up with the possibility that the real issue which should be approached was the system itself. While it was so hard to convert from one system to another, one should instead replace one system with the other so that conversion becomes superfluous. He argued that the whole basis for the analysis framework should be exchanged with the basis of the design community. A paper was released on the subject by Hughes et al. [14] and many others were soon to follow (Bazilevs et al. 2006 [3] among others, Cottrell et al. 2007 [9]).

We need to stress the fact that this is not exclusively an analysis endeavor. Even if the design community represents a much larger market power, which will make it harder to enforce any major changes in that industry. Recent estimates is considering CAD to be a \$5-\$10 billion industry, while the CAE industry is only estimated to \$1-\$2 billions [14]. Changes however needs be done on both sides of these communities. While the analysis would need to work with different geometric constructs and basis functions, the design community would need to create models better suited for analysis. In short, the interaction between analysis and design should flow more naturally.

1.5 T-splines

In later years there has been research on a particular new technique which is called T-splines [20],[19]. It seems that they contain multiple properties which would make them excellent in the union of the CAD and CAE communities. There are several severe problems in the conversion from a CAD-file surface to a model suitable for analysis. One of these is that when designing surfaces in any CAD program, it is not necessary to clean up all gaps. It is accepted to have small gaps as long as they are visually insignificant. This is resulting in a model with topological holes in it. Even if there are small holes, a FEM analysis would require a gap-free, or so-called watertight model. T-splines give a solution to this problem. Another problem is that designers frequently use trimmed surfaces. Trimmed surfaces allow for much more powerful manipulation of the geometry, but they pose a serious problem when converting to an analysis suitable model. This problem also has its solution in T-splines. At last there is a huge problem that CAD designers are creating surfaces, while analysts usually want solids. It has yet to be developed a good surface to solid algorithm, but there are reason to believe that T-splines might provide a solution to this as well. There is ongoing research by among others T.J.R Hughes and Mike Scott on this topic, but no results have been published yet.

Not only are T-splines superior for making the transition from CAD to CAE seemingly more painless, but it also has superior properties from an exclusive design perspective or analysis perspective. It allows designers to manipulate complex models with far less control points than required by traditional non-uniform rational B-splines (NURBS). Sederberg [20] reports that you could model a spline human head object with only 1109 control points over the 4712 which was required with the traditional NURBS approach. From an analyst's point of view, you could allow for true local refinement since you are no longer required to work on tensor product structures.

T-splines seem to be a very appealing way of unifying the CAD and CAE communities, not only for its ease of integrating into existing CAD-programs, but also for its powerful properties within both fields. Moreover, T-splines are completely backward compatible. Both in an analysis and design setting. They are a superset of NURBS, which is by far the dominant technology in CAD-systems today, and NURBS are again a superset of other classical FEM basis functions.

1.6 This thesis

In this thesis we will assume that the reader is already familiar with the finite element method and other classical numerical techniques and results. We will give a detailed introduction of splines, which is the new concept entering the numerical analysis. It is also the choice of spline basis functions, which are defining the isogeometric paradigm.

We will first give a detailed introduction to B-splines and their rational counterpart NURBS, before introducing T-splines, which will in many ways constitute the core of this thesis. In Chapter 4 we will highlight a few of the major differences one encounters when creating a FEM solver based on splines as opposed to more traditional choices of basis functions. After this, we will present two popular test problems in Chapter 5 along with the FEM framework which we will be using. The algorithms presented have been implemented and the numerical results are presented in chapter 6. Chapter 7 will sum up and present our conclusions and suggest possibilities for future work. Finally, in the appendix we will give a slightly more technical introduction into the most frequent practical problems encountered when programming splines.

Chapter 2

B-Splines

We will begin by introducing B-splines. As will later be apparent, both NURBS, PB-splines and T-splines are built up from B-splines, so understanding these will be paramount to further explaination.

B-Splines are piecewise polynomial functions like most splines. They are defined over a series of connected intervals in which they are regular polynomials with all the important properties polynomials possess. They are differentiable, continuous and fairly easy to evaluate. At the interval boundaries however, they have limited continuity and thus also differentiability. All of these properties are equal to those of other splines, like the natural cubic splines etc. However there is one very apperent difference, which is that B-splines are *not* interpolating. This means that the values of the control points, which will be explained below, have little direct interpetation, but must be postprocessed to aquire useful information. This is one of the notable properties of B-splines, and something we will have to deal with.

The reason for using B-spline basis functions is, as already mentioned, because these are the basis used in most CAD systems. In these systems they are used for representing the geometry of either curves or surfaces. While it is possible to create spline solids as well, this is rarely or never done in CAD systems. The extension of the theory from surface to solids however is a straightforward process, and we will here only present spline curves and spline surfaces. We will later use these same basis functions to describe the solution of our differential equation, but for the present time, we will think of this only in terms of geometry.

2.1 Knot vectors

Let the B-spline consist of n piecewise polynomial basis functions, and let p define the degree of these polynomials. The B-spline will then be defined by a set of basis functions who in turn are defined by a set of knots ξ_i which will correspond to the boundaries in between the different polynomials.

Definition 2.1.1 The *knots* is a set of nondecreasing real values $\xi_1, \xi_2, \xi_3, ...\xi_{n+p+1}$.

Together these form the *knot vector* Ξ.

Note that it is only the relative difference between the knots which will play any importance when it comes to defining the B-spline. This means that we can add any constant to all the knot values, and it will generate the same spline in the end. Likewise we can multiply all the knots by a constant factor, and the spline will remain unchanged. The number of knots in the knot vector will be equal to $n+p+1$, where n is the number of basis functions we want to create and p is their polynomial degree. There seems to be some disagreement in the community whether to speak of *order* or *degree* of polynomials. We will in this thesis not distinguish between the two and in all cases, both of these will refer to the highest exponent of the polynomial, i.e. $f(x) = ax^2 + bx + c$ will be a second order polynomial, as well as a second degree polynomial. Even if it has three degrees of freedom a, b and c.

Note that the knots need only be nondecreasing. This means that one can have multiple knots of the same value, we can then talk of the multiplicity of the knot. By increasing the multiplicity we are also decreasing the continuity of a B-spline. In general, a B-spline will have C^∞-continuity at all points, since it consists of regular polynomials, except at the knots, where it has limited continuity. At the knots, the spline will have C^{p-m}-continuity, where p is the polynomial degree of the spline, and m is the multiplicity of the knot. Note that in the case $m = p$ the spline will only be C^0 which means that the spline will not even have a continuous first derivative, and it is thus possible to create sharp corners in the spline curve. In the case $m = p+1$ the continuity will be C^{-1} and the spline will be discontinuous. Since it is included in the spline definition that it should be continuous, it will only be possible to have knots of multiplicity $p+1$ at two points, namely the start- and end-point of the spline. It is customary to use this as the definition of the start and end of the spline, and create the necessary amount of knots to satisfy this.

2.2 B(asis) splines

Armed with the knot vector we will proceed with the creation of the basis functions themselves.

Definition 2.2.1 *The B-spline basis functions* are defined recursively by

$$N_{i,p}(\xi) = \frac{\xi - \xi_i}{\xi_{i+p} - \xi_i} N_{i,p-1}(\xi) + \frac{\xi_{i+p+1} - \xi}{\xi_{i+p+1} - \xi_{i+1}} N_{i+1,p-1}(\xi), \tag{2.1}$$

and,

$$N_{i,0}(\xi) = \begin{cases} 1 & \text{if} \quad \xi \in [\xi_i, \xi_{i+1}) \\ 0 & \text{else} \end{cases} \tag{2.2}$$

where $N_{i,p}$ are the i'th basis function of order p, $i \in [1, n]$.

We also define the fraction in front of the basis functions to be zero in the case of the denominator being zero. That is

$$\begin{aligned} \frac{\xi - \xi_i}{\xi_{i+p} - \xi_i} &\equiv 0 \qquad \text{if } \xi_{i+p} - \xi_i = 0 \\ \frac{\xi_{i+p+1} - \xi}{\xi_{i+p+1} - \xi_{i+1}} &\equiv 0 \qquad \text{if } \xi_{i+p+1} - \xi_{i+1} = 0. \end{aligned}$$

We see that there are exactly n basis functions. This is a direct consequence of the knot vector consisting of $n + p + 1$ knots. If we increase the number of elements in the knot vector, we will also increase the number of basis functions.

2.3 An illustrative example

To get a better grasp of the basis functions we will give an illustrated example on how to create the second order functions corresponding to the knot vector

$$\Xi = [0, 0, 0, 1, 2, 3, 3, 4, 4, 4]. \tag{2.3}$$

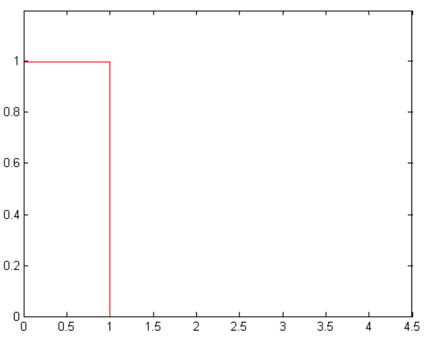
We start off by creating the zero-order functions. As is seen by (2.2), this will only be a partition of the parameter space into piecewise constant functions.

$$\begin{aligned} N_{1,0} &= 0 \\ N_{2,0} &= 0 \\ N_{3,0} &= \begin{cases} 1 & \text{if } \xi \in [0,1) \\ 0 & \text{otherwise} \end{cases} \\ N_{4,0} &= \begin{cases} 1 & \text{if } \xi \in [1,2) \\ 0 & \text{otherwise} \end{cases} \\ N_{5,0} &= \begin{cases} 1 & \text{if } \xi \in [2,3) \\ 0 & \text{otherwise} \end{cases} \\ N_{6,0} &= 0 \\ N_{7,0} &= \begin{cases} 1 & \text{if } \xi \in [3,4) \\ 0 & \text{otherwise} \end{cases} \\ N_{8,0} &= 0 \\ N_{9,0} &= 0 \end{aligned} \tag{2.4}$$

The first order functions are then created from each pair of consecutive zero-order basis function by blending these with linear functions, i.e. the creation of $N_{3,1}$ would be given from (2.1) as

$$N_{3,1}(\xi) = \frac{\xi - 0}{1 - 0} N_{3,0}(\xi) + \frac{2 - \xi}{2 - 1} N_{4,0}(\xi). \tag{2.5}$$

We see that the function $\xi \to \frac{\xi - 0}{1 - 0}$ is a linear function equal to zero at the beginning of the nonzero domain of $N_{3,0}$ and ending equal to one at the end of the nonzero domain


(a) The basis function $N_{3,0}$

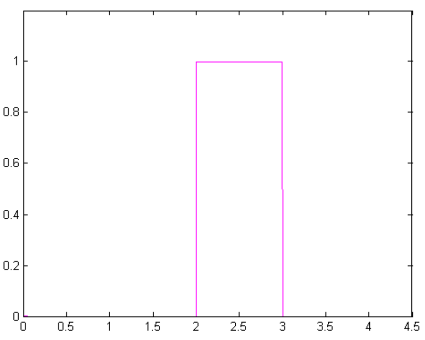

(b) The basis function $N_{5,0}$

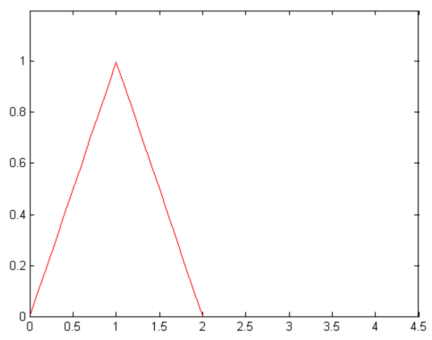
Figure 2.1: The constant basis functions

of $N_{3,0}$. Likewise, $\xi \to \frac{2-\xi}{2-1}$ is a linear function decreasing from 1 at the start of $N_{4,0}$'s domain and zero at the end. Since the zero-order functions are all defined piecewise, the first-order function will also be defined piecewise. When inserting (2.4) into (2.5) we get $N_{3,1}$ as written in (2.7). The computation of $N_{4,1}$ is completely analogous, however in the case of $N_{2,1}$ we see that the denominator is zero in the first term

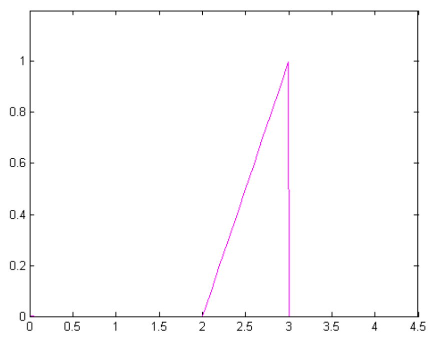
$$N_{2,1}(\xi) = \frac{\xi - 0}{0 - 0} N_{2,0}(\xi) + \frac{1-\xi}{1-0} N_{3,0}(\xi). \tag{2.6}$$

By (2.3) this term vanishes, and we are left with the linear decreasing function in $\xi \in [0,1)$.

To sum up, all the first order functions will be given as in (2.7)



(a) The basis function $N_{3,1}$



(b) $N_{5,1}$ - this is discontinuous at the multiple knot $\xi = 3$

Figure 2.2: First order basis functions

$$\begin{aligned}
N_{1,1} &= 0 \\
N_{2,1} &= \begin{cases} 1-\xi & \text{if } \xi \in [0,1) \\ 0 & \text{otherwise} \end{cases} \\
N_{3,1} &= \begin{cases} \xi & \text{if } \xi \in [0,1) \\ 2-\xi & \text{if } \xi \in [1,2) \\ 0 & \text{otherwise} \end{cases} \\
N_{4,1} &= \begin{cases} \xi-1 & \text{if } \xi \in [1,2) \\ 3-\xi & \text{if } \xi \in [2,3) \\ 0 & \text{otherwise} \end{cases} \\
N_{5,1} &= \begin{cases} \xi-2 & \text{if } \xi \in [2,3) \\ 0 & \text{otherwise} \end{cases} \\
N_{6,1} &= \begin{cases} 4-\xi & \text{if } \xi \in [3,4) \\ 0 & \text{otherwise} \end{cases} \\
N_{7,1} &= \begin{cases} \xi-3 & \text{if } \xi \in [3,4) \\ 0 & \text{otherwise} \end{cases} \\
N_{8,1} &= 0
\end{aligned} \tag{2.7}$$

Now, note that the basis functions are *discontinuous* at $\xi = 3$. This is due to the fact that in our knot vector (2.3), there is a knot of multiplicity two at that point, which means that the first order functions will be of continuity $p - m = -1$. If we were to create only the first order functions, then this would mean that the knot vector would end at this point and no more basis functions would be created. However, since we are only using this as an intermediate step in creating the second-order functions, which at this point will be C^0, we continue. For anyone experienced with the finite element method, these functions will probably look familiar. It is true that they look familiar, and the zero- and first order functions are identical to those used in "classical" linear FEM. When we extend the basis to second order, this will however change and the functions will not be the same anymore.

When we in our next step create the second order functions, these are created in the exact same manner as before. The basis functions are created by blending two consecutive first-order functions with two linear functions, and then added. Thus, for creating $N_{3,2}$ we have

$$N_{3,2}(\xi) = \frac{\xi - 0}{2 - 0} N_{3,1}(\xi) + \frac{3 - \xi}{3 - 1} N_{4,1}(\xi), \tag{2.8}$$

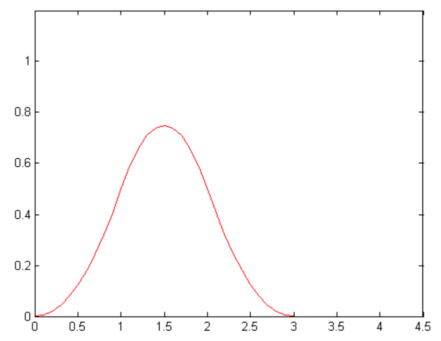
Which inserted from (2.7) gives

$$N_{3,2}(\xi) = \begin{cases} \frac{1}{2}\xi^2 & \text{if } \xi \in [0,1) \\ \frac{\xi}{2}(3-2\xi) & \text{if } \xi \in [1,2) \\ \frac{1}{2}(3-\xi)(2-\xi) & \text{if } \xi \in [2,3) \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$

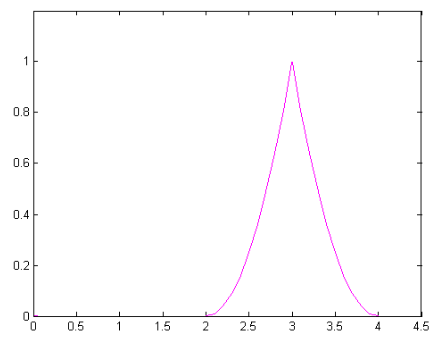
When calculating the rest of the basis functions they will be given as

$$\begin{aligned}
N_{1,2} &= \begin{cases} (1-\xi)^2 & \text{if } \xi \in [0,1) \\ 0 & \text{otherwise} \end{cases} \\
N_{2,2} &= \frac{1}{2}\begin{cases} 2\xi(3-2\xi) & \text{if } \xi \in [0,1) \\ (2-\xi)^2 & \text{if } \xi \in [1,2) \\ 0 & \text{otherwise} \end{cases} \\
N_{3,2} &= \frac{1}{2}\begin{cases} \xi^2 & \text{if } \xi \in [0,1) \\ \xi(3-2\xi) & \text{if } \xi \in [1,2) \\ (3-\xi)(2-\xi) & \text{if } \xi \in [2,3) \\ 0 & \text{otherwise} \end{cases} \\
N_{4,2} &= \frac{1}{2}\begin{cases} (\xi-1)^2 & \text{if } \xi \in [1,2) \\ 2(3-\xi)(2\xi-3) & \text{if } \xi \in [2,3) \\ 0 & \text{otherwise} \end{cases} \\
N_{5,2} &= \begin{cases} (\xi-2)^2 & \text{if } \xi \in [2,3) \\ (4-\xi)^2 & \text{if } \xi \in [3,4) \\ 0 & \text{otherwise} \end{cases} \\
N_{6,2} &= \begin{cases} 2(4-\xi)(\xi-3) & \text{if } \xi \in [3,4) \\ 0 & \text{otherwise} \end{cases} \\
N_{7,2} &= \begin{cases} (\xi-3)^2 & \text{if } \xi \in [3,4) \\ 0 & \text{otherwise} \end{cases} \\
N_{8,2} &= 0
\end{aligned} \tag{2.10}$$

As is seen from (2.9) we have that the second order basis functions have nonzero values of up to three knot intervals. We will refer to the values in which the basis function is nonzero as the support of the function.



(a) The basis function $N_{3,2}$



(b) $N_{5,2}$ - this is C^0 at the multiple knot $\xi = 3$

Figure 2.3: Second order basis functions

Now as seen from the example it is apparent that it is quite *possible* to create the exact polynomials which make up the basis functions, but, from a computational point of view this is highly inefficient and we will instead be working from the recursive definition given by (2.1). Even if one takes into account the fact that the creation of the functions would only be calculated once, the evaluation of the basis functions would still not compare to a dynamic programming approach. For full details on a computational efficient algorithm for the evaluation of the basis functions see appendix A.

2.4 General properties

As seen in the example above, the supports of the basis functions are growing with increasing polynomial degree. In fact, the support will cross exactly $p+2$ knots. Note however that some of these knots may be equal and thus have knot multiplicity greater than one. In these cases, the support will not go over $p+1$ knot spans, which is the maximum support it can have.

In addition, as the name suggest, the basis functions will indeed form a *basis*. That is, they are all linearly independent.

The basis forms a partition of unity meaning that the sum of them will always equal one for any given p, i.e.

$$\sum_{i=0}^{n-1} N_{i,p}(\xi) = 1, \quad \forall p \in \mathbb{Z}^+,\ \xi \in \mathbb{R} \tag{2.11}$$

2.5 From basis to curve

Now that we are armed with the basis functions, it is time to move them into the actual space which we will be looking at. This might be one of several spaces, but in light of

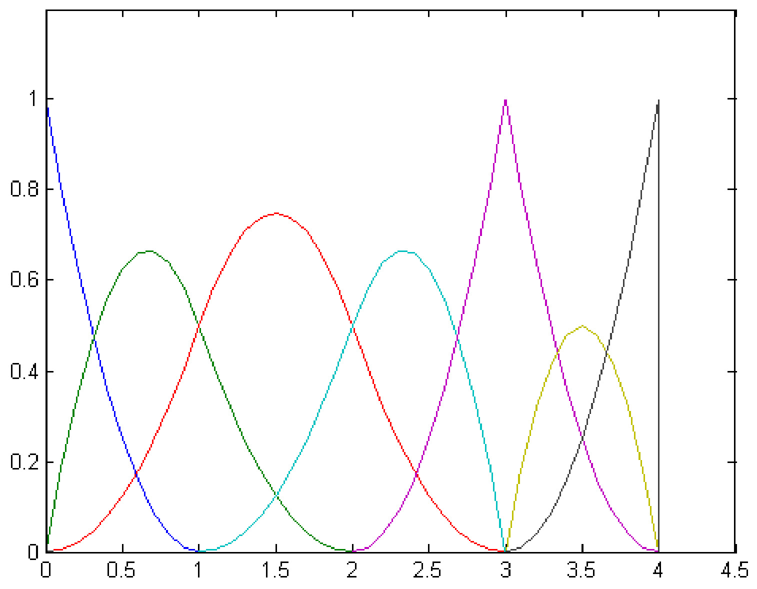


Figure 2.4: All second-order basis functions corresponding to $\Xi = [0, 0, 0, 1, 2, 3, 3, 4, 4, 4]$

curve and surface representations, it is most common to use $\mathbb{R}^2$ or $\mathbb{R}^3$. The mapping from the parameter space to the physical space is then given by

$$\boldsymbol{C} = \sum_{i=1}^{n} N_{i,p}(\xi)\boldsymbol{B}_i, \tag{2.12}$$

where the $\boldsymbol{B}_i$'s are the control points of the spline. They may be points in $\mathbb{R}, \mathbb{R}^2$ or $\mathbb{R}^3$. It is here important to stress the fact that the control points are in general *not* interpolating. The curve will not go through the control points, neither will the knots themselves correspond to any particular control point. It is hard to put any direct geometric link between the control points and the basis functions. While it is obvious from an algebraic point of view that control point i is the weighting of basis function i from (2.12), this is much harder to pin-point from a geometric perspective. To illustrate this, we have created a second-order curve from the above example with the knot vector given in (2.3). In this example we are using control points in $\mathbb{R}^2$. The control points themselves are points in the physical space and can be directly plotted in the figure. To get the knots, we have to evaluate these ξ-values using (2.12). The control points are plotted as squares, and the knots as circles in figure 2.5

While it is hard to pin-point a geometric link between the control points and the knots or basis functions, there is however some properties of the B-splines that allows an intuitive interpretation. If we draw a control polygon given by the ordered set of control

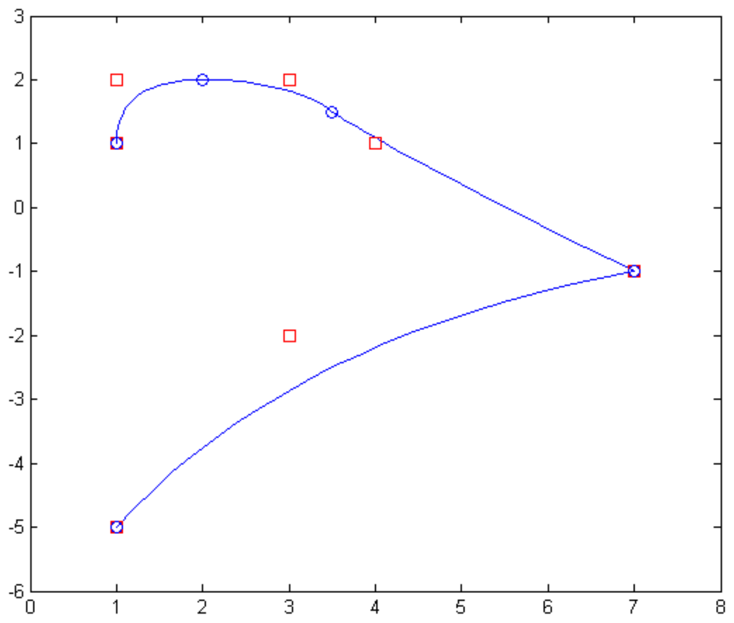


Figure 2.5: An example B-Spline curve with knots plotted as circles and control points as squares

points $\boldsymbol{B}_i$, then the curve $\boldsymbol{C}$ will lie entirely within this polygon. Other than this, the local support of the basis functions ensures that only a small part of the curve change if one were to change the location of one of the control points. Moreover, the part of the curve which will change is the part closest to the control point. This makes the control points very intuitively to work with from a design perspective. By dragging the control points themselves, you are effectively locally manipulating the curve. This is also one of the reasons that they have had such huge success within the CAD community. From an analysis perspective however, the non-existent link between the parameter space of the basis functions and the knot vector, and the actual world geometry makes it them slightly tiresome to work with.

2.6 A 2D tensor product

The extension from curve to surface is straightforward. Where we before had one knot vector $\Xi = [\xi_1, ..., \xi_{n_1+p_1+1}]$ we will now have an additional $\mathcal{H} = [\eta_1, ..., \eta_{n_2+p_2+1}]$. This will then define another set of linearly independent basis functions $N_{j,p_2}(\eta)$ according to (2.1). The two dimensional basis functions will then be given as the tensor product of any pair (i, j) of these functions, i.e.

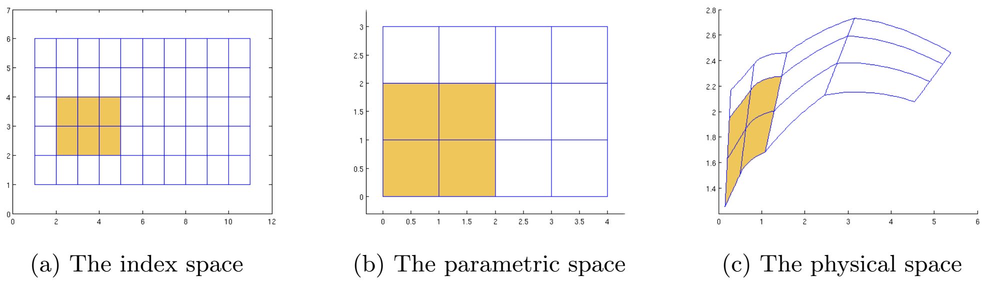

(a) The index space (b) The parametric space (c) The physical space

Figure 2.6: A B-spline and how it appears in different spaces. The support of the basis function $N_{2,2}(\xi)N_{2,1}(\eta)$ is shaded

$$N_{i,j,p_1,p_2}(\xi,\eta) = N_{i,p_1}(\xi)N_{j,p_2}(\eta)\ , \quad \begin{array}{ll} i & \in \{1,...,n_1\} \\ j & \in \{1,...,n_2\} \end{array} . \tag{2.13}$$

p_1 and p_2 need not be equal and are defining the polynomial degrees in the parameter direction ξ and η respectively. Once again, we will require the first and last of the control points in Ξ to have multiplicity p_1+1 and the first and last of $\mathcal{H}$ to have multiplicity p_2+1. The parameter domain defining the surface will then be the rectangle $(\xi,\eta) \in [\xi_1, \xi_{n_1+p_1+1}] \times [\eta_1, \eta_{n_2+p_2+1}]$.

We will also now need a net of control points $\boldsymbol{B}_{i,j}$ (which may be in $\mathbb{R}^2$ or $\mathbb{R}^3$). Which in turn produces the surface by

$$\boldsymbol{C}(\xi,\eta) = \sum_{i,j} N_{i,p_1}(\xi)N_{j,p_2}(\eta)\boldsymbol{B}_{i,j} \tag{2.14}$$

Now to get a better understanding of what is going on we are going to have a closer look on the interaction between the knots, basis functions and the spline surface when dealing with the two-dimensional case. We will also introduce the index space which will act as a second layer of abstraction in addition to the parameter space. The index-space is simply the area defined by integers 1 to n_1+p_1+1 in the i direction and 1 to n_2+p_2+1 in the j direction. This will consist of a net of index knot points for all pairs of (i,j). The index space is illustrative for analyzing the support of the basis functions, which can easily be read in this space, but are less apparent in the parameter space. It is also paramount to the understanding of T-splines which will be introduced later. The concept is best illustrated in figure 2.6 where we see the index space, parameter space and the physical space of a B-spline surface corresponding to the knot vectors $\Xi = [0,0,0,1,2,2,3,3,4,4,4]$, $\mathcal{H} = [0,0,1,2,3,3]$ and the degrees $p_1 = 2$, $p_2 = 1$.

Since we know that the support of any basis function cross exactly 4 indices in the i direction, and 3 indices in the j direction we can immediately mark the support of any basis function in the index domain. Due to some knots having higher multiplicity this is not readily apparent in the parameter domain. The support of basis function $(2,2)$ is marked in figure 2.6 and as can be seen, the support is not over a 3 by 2 knot span which is the maximum support of any $(p_1, p_2) = (2, 1)$ B-spline basis function.

Again we want to stress the absence of interaction between control points and knots. Neither the "nodes" in the index space nor the parameter space are corresponding to any particular control point, and referring to these points in such a manner could cause confusion. The concept of node will be reserved for classical FEM, and we will instead name the intersections in the index-space index knot, and the intersections in the parameter space for knot.

2.7 Analysis on splines

Up until this point we have seen the splines from the perspective of their most frequent use: geometric object description. However, when using them for analysis purposes we will not only need the basis functions themselves, but also their derivative. It should not be surprising in light of the recursive definition that the derivatives can be described by a recursive relationship as well. They are given by

$$\frac{d}{d\xi}N_{i,p}(\xi) = \frac{p}{\xi_{i+p}-\xi_i}N_{i,p-1}(\xi) - \frac{p}{\xi_{i+p+1}-\xi_{i+1}}N_{i+1,p-1}(\xi). \tag{2.15}$$

This can readily by generalized to higher order derivatives by implicit derivation, that is differentiate each side by ξ and rearrange the terms

$$\frac{d^k}{d\xi^k}N_{i,p}(\xi) = \frac{p!}{(p-k)!}\sum_{j=0}^{k}\alpha_{k,j}N_{i+j,p-k}(\xi), \tag{2.16}$$

with

$$\begin{aligned}
\alpha_{0,0} &= 1 \\
\alpha_{k,0} &= \frac{\alpha_{k-1,0}}{\xi_{i+p-k+1}-\xi_i}, \\
\alpha_{k,j} &= \frac{\alpha_{k-1,j}-\alpha_{k-1,j-1}}{\xi_{i+p+j-k+1}-\xi_{i+j}} \quad j = 1, ..., k-1, \\
\alpha_{k,k} &= \frac{-\alpha_{k-1,k-1}}{\xi_{i+p+1}-\xi_{i+k}}.
\end{aligned}$$

As it stands it is quite possible to refine the solution space of any particular B-spline. Simply by inserting a single knot at an arbitrary ξ, we have inserted a new basis function. However if we wish to preserve the physical geometric object which is described by the spline we will have to do this in a more sophisticated way. The process of knot insertion is luckily an exact science and it is possible to insert knots at arbitrary positions while still maintaining the geometric object untouched. This is done by a splitting algorithm which basically splits one basis function (with corresponding control point) into two and afterwards slightly alters the position of up to p control points. For full details on the derivation of the knot insertion algorithm see [8]. We will stick with the notation established in Hughes et. al [14] and refer to this refining process as h-refinement.

It is also possible to elevate the polynomial degree of the curve. Note however that the degree of a B-spline curve is a global property and unlike knot insertion which might be used to locally increase the number of basis functions, the elevation of the polynomial order will affect the entire curve. Like the knot insertion, it is possible to increase the polynomial degree without altering the shape of the geometric object in the physical space. The exact algorithm for raising the order of a B-spline can be found in [13]. We will refer to this type of refinement as p-refinement.

The refining algorithms are not commutative. That is doing h-refinement first and then p-refinement will yield different results than doing p-refinement first followed by h-refinement. Hughes [15] is introducing a special way of ordering the refinement process of p- and h-refinement which has superior properties when it comes to among other things continuity and number of degrees of freedom. He dubs this refinement process k-refinement, but we will not be discussing this technique in this thesis.

2.8 NURBS

While the B-spline has seen many uses, it has severe flaws. Among these is the fact that it is incapable of representing conic sections, i.e. circle, parabola and hyperbola, exactly. This is due to the fact that none of these can be described as piecewise polynomials. While it is possible to approximate a circle, one would obviously expect the possibility of creating such an elementary primitive from an advanced technique like splines. The result of this pursuit was the non-uniform rational B-splines (NURBS). While we have already discussed non-uniform B-splines (referring to the knot vector being non-uniform), the new aspect which comes into the picture is that the splines will no longer be piecewise polynomials, but rather piecewise *rational* polynomials.

2.8.1 Geometric perspective

NURBS will be built upon the already existing knowledge-base of the B-splines. In fact, a NURBS curve will be the shadow of a plain B-spline curve projected onto a constant plane. Let us consider a B-spline curve in $\mathbb{R}^{(d+1)}$, which is simply defined by letting the control points $\boldsymbol{B}_i$ having $d+1$ components. This curve will then be (perspective) projected onto the plane $z = 1$ by straight lines going through the origin. This is a straightforward equation to solve by using the fact that the triangles will be similar (figure 2.7). The result is that the projected control points $\boldsymbol{B}_i$ are related to the B-spline control points $\boldsymbol{B}_i^w$ by

$$\begin{aligned}
(\boldsymbol{B}_i)_j &= \frac{(\boldsymbol{B}_i^w)_j}{w_i}, \qquad j = 1, ..., d \\
w_i &= (\boldsymbol{B}_i^w)_{d+1}
\end{aligned} \tag{2.17}$$

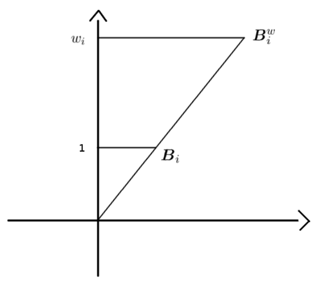


Figure 2.7: NURBS are created by projecting a B-spline into the plane $w = 1$

where $(\boldsymbol{B})_j$ is the j'th component of the vector $\boldsymbol{B}_i$ and w_i is referred to as the i'th weight.

Realizing that NURBS are in fact only regular B-splines projected to a space of one dimension less, is very instructive since it will ease many of the preceding results. Many of the results obtained from the investigation of B-splines follow directly when looking at NURBS. The knot insertion algorithm consists of three steps, project the curve out into $d+1$ dimensions, use regular B-spline knot insertion and project it back into d dimensions afterwards. This algorithm will come very natural and it is because of this that we point it out. While it is illustrative and useful to realize this fact, it becomes quite tedious if we are to create the $d+1$-dimensional B-spline which gives the desired NURBS-curve under projection. This is why we seldom use this in practice when creating or manipulating NURBS. Instead we will use templates or other tools for generating the geometry.

2.8.2 Algebraic perspective

The NURBS are, as already mentioned, piecewise rational polynomial functions, and their basis functions are given as

$$R_{i,p}(\xi) = \frac{N_{i,p}(\xi)w_i}{W(\xi)} = \frac{N_{i,p}(\xi)w_i}{\sum_{\hat{i}=1}^{n_1} N_{\hat{i},p}(\xi)w_{\hat{i}}} \tag{2.18}$$

and for the NURBS surface

$$R_{i,j,p_1,p_2}(\xi,\eta) = \frac{N_{i,p_1}(\xi)N_{j,p_2}(\eta)w_{i,j}}{\sum_{\hat{i}=1}^{n_1}\sum_{\hat{j}=1}^{n_2} N_{\hat{i},p_1}(\xi)N_{\hat{j},p_2}(\eta)w_{\hat{i},\hat{j}}}. \tag{2.19}$$

While the basis functions have changed significantly, the mapping from the parameter space to the physical space takes the same form, i.e.

$$\boldsymbol{C}(\xi) = \sum_{i=1}^{n} R_{i,p}(\xi)\boldsymbol{B}_i \tag{2.20}$$

$$\boldsymbol{C}(\xi,\eta) = \sum_{i=1}^{n_1}\sum_{j=1}^{n_2} R_{i,j,p_1,p_2}(\xi,\eta)\boldsymbol{B}_{i,j} \tag{2.21}$$

We will not go more into detail on the NURBS. For our purposes it is sufficient to know that we can create a rational B-spline by adding a weight to each control point, and altering the way that we are evaluating the basis functions. This will extend to T-splines as well. We will discuss the T-splines as if they were an extension of the B-splines, but conversion to rational T-splines is a straightforward process.

Chapter 3

T-Splines

The problem with B-splines and NURBS is that they are formulated as tensor products. This means that refinement will cause an entire new row or column of knots to be formed. While you get refinement around a local point, it will also cause a lot of refinement to take place in other areas of the mesh. This is illustrated in figure 3.1 where we have recursively refined the lower right corner. Ideally we do not want any more knots to appear in the upper right and lower left, but with B-splines and NURBS, this is unavoidable. To achive true local refinement we will have to introduce some new structure to the mesh which is not formed by global tensor products. This is where T-splines will enter the picture. T-splines was first introduced by Sederberg et al. (2003) [20] and has, like B-splines and NURBS, primarily been used for visualization purposes. In later years it have however seen its introduction into FEM analysis as well [10], [4]. We will in this section give an introduction to T-splines and most importantly the refining algorithm. We will also discuss the impact T-splines may have in a finite element setting.

3.1 PB-splines

We will start the introduction of T-splines by describing a more general type of splines, the point B-spline (PB-spline). This is a B-spline, but one that is formed by choosing a set of independent points and knot vectors. As we remember from chapter 2, the B-splines basis functions were only defined by the knot vectors. Given a single knot vector we could calculate the entire set of corresponding basis functions in the parameter space. If one were to look closer at this statement, we see that the basis functions will not be dependent on the whole knot vector, they will only be dependent on a subset of the knot-vector. For any basis function of order p, we will have exactly $p+1$ knots which will influence that particular basis function. These knots will be a connected subset of the entire knot vector. This all follows from the definition of the basis functions in (2.1). As an example, let us look at a particular knot vector, $\Xi = [0,0,0,1,2,3,3,4,4,4]$ for a second order B-spline. Corresponding to this vector we will have the basis functions $N_{1,2}, N_{2,2}, ..., N_{7,2}$. The function $N_{1,2}$ is only dependent on the first four elements of the knot vector, i.e. $\Xi_1 = [0,0,0,1]$. The function $N_{2,2}$ will only be dependent on

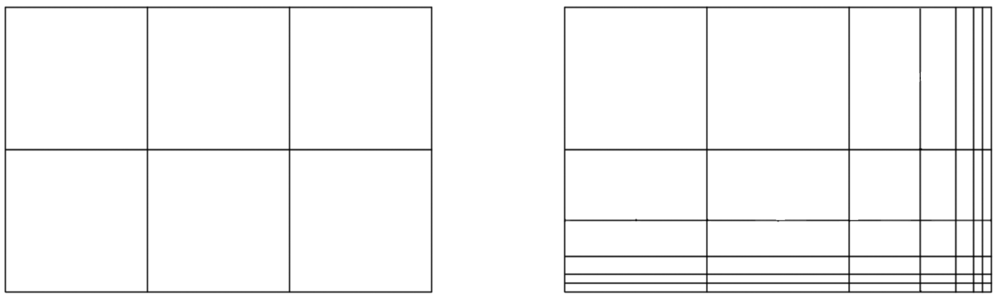

(a) Mesh without refinement (b) Tensor product refinement

Figure 3.1: Tensor product problems

$\Xi_2 = [0, 0, 1, 2]$. The function $N_{3,2}$ will only depend on $\Xi_3 = [0, 1, 2, 3]$, etc.

Now, if one were to break this dependence free from the knot vector itself, then it is clear that we can create any single B-spline basis function from a knot-vector of size $p+2$. If these knot vectors were allowed to be picked independently, instead of as a continuous subset of a global knot vector, then we will get something that we call PB-splines.

Definition 3.1.1 *A PB-spline is a collection of blending functions where each blending function N_i is a B-spline basis function generated from a set of local nondecreasing knot vectors Ξ_i and $\mathcal{H}_i$ whose elements are independent of any other local knot vectors Ξ_j or $\mathcal{H}_j$, $i \neq j$.*

Note that we say "blending" function and not "basis" function, and this is because the set of functions that form the PB-spline, will in general *not* be linearly independent. While this was the case for B-splines, this property is lost while dealing with such general structures. This is obvious since there is no restrictions in the definition from creating two blending functions with identical knot vectors. The functions would then be identical as well, and obviously dependent. This particular example is not possible when dealing with B-splines since the global knot vector Ξ is non-decreasing we would have to have two equal local knot-vectors Ξ_i and Ξ_j where every element of Ξ_i was equal. This would then cause the multiplicity of that knot to be $p + 2$, but no knot in Ξ except the end points have multiplicity greater than $p + 1$ since this is the definition of the start and end of the knot-vector.

We have now introduced the concept of *local* knot vectors and this will play a very important role in the following discussion on T-splines.

3.2 T-splines

While PB-splines are a set of blending functions with arbitrary knot vectors associated with each blending function, T-splines seem to add more structure to this definition.

This is done by providing an algorithm for generating the knot vectors for any particular function in the domain. While B-splines had a well-defined way of extracting the local knot vectors from the global, and the PB-spline required that every local knot vector was defined by the user, the T-spline will offer a third option when it comes to generating the local knot vector.

Obviously talking about T-splines in one dimension, i.e. T-spline *curves* is meaningless since you could not have any T-joints in one dimension. We will therefore restrict ourselves to only talking about T-spline surfaces, in which we have two axes in the index domain i and j, and two axes in the parametric domain ξ and η. That is not to say that T-splines are restricted to this. Bazilevs et al. [4] have done some work on describing T-spline solids.

But before we describe the algorithm for local knot vector extraction, we will need to establish some more notation.

Definition 3.2.1 A T-mesh is a collection of points $\mathcal{P}$, ξ-edges $\mathcal{E}_\xi$ and η-edges $\mathcal{E}_\eta$ in the index-domain, where each edge is required to be horizontal (ξ-line) or vertical (η-line).

Note that this definition allows for a very general T-mesh. There is no restriction on the faces being rectangles, or the connection of the edges. Points can exist in T-joints (figure 3.2), L-joints or I-joints (figure 3.3). It is even allowed for "loose" points with no connecting lines. In this thesis however, we will not consider such general constructs, and the only joints which will appear as a result of the T-spline refining algorithm is L-joints and T-joints.

Definition 3.2.2 A T-spline is a T-mesh associated with two global knot vectors Ξ and $\mathcal{H}$ with a number of elements equal to the maximum index in respectively the i and j direction of the T-mesh and a corresponding control point $\boldsymbol{B}_k$ for each point $P_k \in \mathcal{P}$.

The number of knots in the knot vector is no longer $p + n + 1$ since we will not have n basis functions in each parametric direction. Instead, we will have a collection of basis functions which will be dictated by the T-mesh. Note in figure 3.2-3.3 that there are legends on the axes. These indicate positions in the global knot vectors for one particular point in the index domain.

Definition 3.2.3 The local index knot vectors $I_k = [i_1, i_2, ..., i_{p_1+2}]$ and $J_k = [j_1, ..., j_{p_2+2}]$ will generate the local knot vectors Ξ_k and $\mathcal{H}_k$ by

$$\Xi_k(I, \Xi) = [\xi_{i_1}, ..., \xi_{i_{p_1+2}}] \tag{3.1}$$

$$\mathcal{H}_k(J, \mathcal{H}) = [\eta_{j_1}, ..., \eta_{j_{p_2+2}}] \tag{3.2}$$

The index knot vectors are strictly not necessary, and most authors on T-splines are describing T-splines without these, but they will make both discussion and implementation easier. It is also instructive to realize the fact that the local index knot vectors "live" in the index domain, while the local knot vectors "live" in the parametric domain.

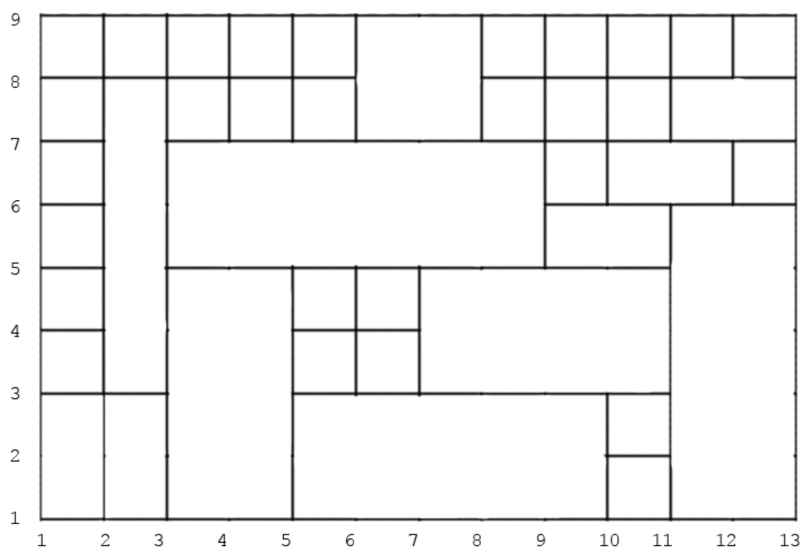


Figure 3.2: A valid T-mesh with only T-joints

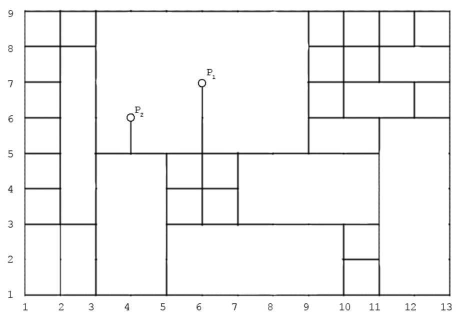
3.3 The knot-vector extraction

The algorithm for extracting the local knot vector from the T-mesh is given in algorithm 1. This will create one basis function for each $P_k \in \mathcal{P}$ of the T-mesh. We will first of all require that the polynomial degrees p_1 and p_2 are odd. This is not in itself a restriction since one can always define the appropriate behavior for even degrees, but this will be unsymmetrical and we will avoid these cases for the time being. What the algorithm in essence is doing is drawing four "rays" out in each direction north, south, east and west to find an appropriate number of intersections. The first index which is to be included is always the point $P = (i, j)$ itself. This is inserted in the middle of the local index knot vectors I and J in line 2-3. In the next section, line 4-8, we are filling the rest of the I vector with the first intersections in the east direction. Line 5 is finding the first point or η-edge which is intersecting the eastward search-ray and stores the i value of this in the I-vector. Then the next intersection will be found and stored until the vector is filled up. The process is repeated for the other three directions: west, south and north.

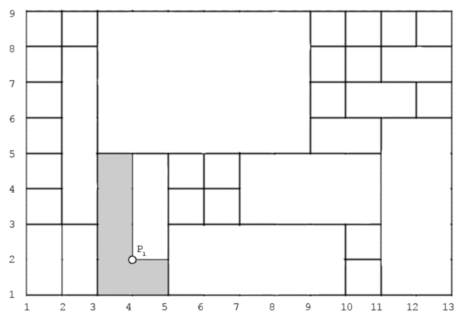
The extraction algorithm for the local knot vectors Ξ_k and $\mathcal{H}_k$ is best illustrated by an example. Let the T-mesh be as indicated in figure 3.4 and let $p_1 = 3$ and $p_2 = 1$ so we have a spline which is cubic in ξ and linear in η. We then know that for each knot-point $P_k = (i, j)$ there will be a corresponding basis function given by the local knot vectors Ξ_k and $\mathcal{H}_k$. These knot vectors will respectively be of size 5 and 3. We then proceed

Algorithm 1 Local knot vector extraction

```
Require: $p_1$ and $p_2$ odd
 1: $P \leftarrow (i, j)$
 2: $I[\frac{p_1+1}{2}] \leftarrow i$
 3: $J[\frac{p_2+1}{2}] \leftarrow j$
 4: for $k = \frac{p_1+1}{2} + 1$ to $p_1$ do
 5:     $P \leftarrow \min_{\hat{i}}\{((\hat{i}, j) \in \mathcal{P}$ or $(\hat{i}, j) \in \mathcal{E}_\eta)$ and $\hat{i} > P_i\}$
 6:     $I[k] \leftarrow P_i$
 7: end for
 8: $P \leftarrow (i, j)$
 9: for $k = \frac{p_1+1}{2} - 1$ to 1 do
10:     $P \leftarrow \min_{\hat{i}}\{((\hat{i}, j) \in \mathcal{P}$ or $(\hat{i}, j) \in \mathcal{E}_\eta)$ and $\hat{i} < P_i\}$
11:     $I[k] \leftarrow P_i$
12: end for
13: $P \leftarrow (i, j)$
14: for $k = \frac{p_2+1}{2} + 1$ to $p_2$ do
15:     $P \leftarrow \min_{\hat{j}}\{((i, \hat{j}) \in \mathcal{P}$ or $(\hat{i}, j) \in \mathcal{E}_\xi)$ and $\hat{j} > P_j\}$
16:     $J[k] \leftarrow P_j$
17: end for
18: $P \leftarrow (i, j)$
19: for $k = \frac{p_2+1}{2} - 1$ to 1 do
20:     $P \leftarrow \min_{\hat{j}}\{((i, \hat{j}) \in \mathcal{P}$ or $(\hat{i}, j) \in \mathcal{E}_\xi)$ and $\hat{j} < P_j\}$
21:     $J[k] \leftarrow P_j$
22: end for
```



(a) A valid T-mesh even if it is creating I-joints



(b) A valid T-mesh with P_1 being an L-joint

Figure 3.3: Exotic T-mesh construction

to find these vectors. Let $P_1 = (10, 2)$ be the index coordinates of one such point. For notational ease, we will describe the way of extracting I and J which in turn will yield the local knot vectors. The middle point of I and J will always be the point itself. That is $I_3 = 10$ and $J_2 = 2$.

$$\begin{aligned} I_{P_1} &= [\cdot, \cdot, 10, \cdot, \cdot] \\ \Xi_{P_1} &= [\cdot, \cdot, \xi_{10}, \cdot, \cdot] \\ J_{P_1} &= [\cdot, 2, \cdot] \\ \mathcal{H}_{P_1} &= [\cdot, \eta_2, \cdot]. \end{aligned}$$

The next step is to extract the rest of the knot vector. This is done by following a straight line in positive i-direction. For each η-line or point intersected we note the i value at that particular point. From figure 3.4 we see that a line from P_1 will intersect a point in (11,2) and a η-line at (13,2). Thus the last two values of I will be respectively 11 and 13, i.e. $I_{P_1} = [\cdot, \cdot, 10, 11, 13]$. To extract the first two indices, we draw a line in negative i-direction noting each intersection, which is at the η-lines going through (5,2) and (3,2). Hence the final index knot vector will be $I_{P_1} = [3, 5, 10, 11, 13]$ with corresponding local knot vector $\Xi_{P_1} = [\xi_3, \xi_5, \xi_{10}, \xi_{11}, \xi_{13}]$. To extract J we will need to form a ray in the positive and negative j-direction, which will intersect an index knot point at respectively (10,3) and (10,1), thus producing $J = [1, 2, 3]$.

$$\begin{aligned} I_{P_1} &= [3, 5, 10, 11, 13] \\ \Xi_{P_1} &= [\xi_3, \xi_5, \xi_{10}, \xi_{11}, \xi_{13}] \\ J_{P_1} &= [1, 2, 3] \\ \mathcal{H}_{P_1} &= [\eta_1, \eta_2, \eta_3] \end{aligned}$$

For the point $P_2 = (9, 7)$ which is also noted in figure 3.4 this will generate the local

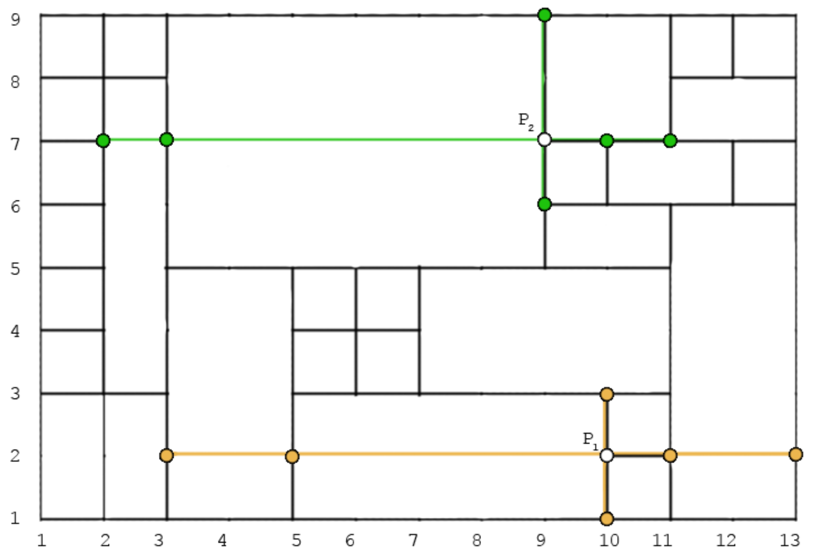


Figure 3.4: An example t-mesh

index- and knot vectors

$$\begin{aligned}
I_{P_2} &= [2, 3, 9, 10, 11] \\
\Xi_{P_2} &= [\xi_2, \xi_3, \xi_9, \xi_{10}, \xi_{11}] \\
J_{P_2} &= [6, 7, 9] \\
\mathcal{H}_{P_2} &= [\eta_6, \eta_7, \eta_9]
\end{aligned}$$

Note that in the case of any of the rays leaving the mesh, it is customary to repeat the last i or j value obtained before leaving the domain.

3.4 Refinement

Inserting new blending functions into a T-spline is trivial. This can be accomplished by just inserting new points into the T-mesh and this will then generate more blending functions. However, this will in general alter the physical shape of the mapped surface. This is fine in many design applications since it is to be understood that the new control points are going to be subject to further alterations anyway. In an analysis setting however, this is unacceptable. Inserting blending functions such that the surface remains unchanged is a slightly more sophisticated technique. It was first introduced by Sederberg (2003) et al. [20], and later improved by Sederberg (2004) et al. [19]. The basic idea behind this insertion algorithm is the continued use of B-spline splitting, but keeping this to a bare minimum. While it is possible in degenerate cases to only insert a single blending function, it seems like it is impossible to do this in a general setting. When inserting one function, that function is usually dependent on the existence of more blending functions which will trigger multiple points being inserted. The number of new points needed was however drastically reduced by the improved algorithm and it is now a very effective and highly local procedure. For full details on the T-spline refinement algorithm see appendix B.

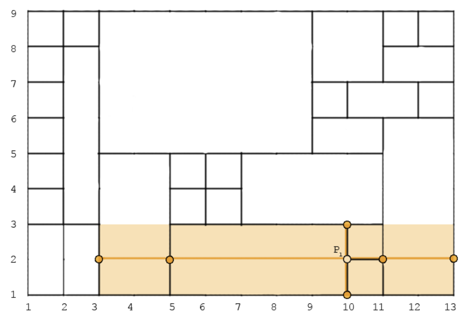
From a T-spline user's point of view it might not be necessary to grasp every detail of the refinement algorithm, it is however important to stress the fact that the refinement scheme is producing more points than requested. You will usually get more points than you bargained for, but it is non the less a local refinement. The new points tend to spread out to the elements in immediate proximity of the ones requested, but with little precise control over exactly where. This should be taken into account when requesting element refinements in an adaptive algorithm scheme. While in classical FEM it is customary to refine the α percent elements with the highest error, this value could be much smaller in a T-spline solver since you would have additional refinements generated automatically from the T-spline refining algorithm. Moreover, these would in general be placed at good positions since they would be close to the original requested refinement, which apparently is a hot spot for errors.

3.5 Element extraction

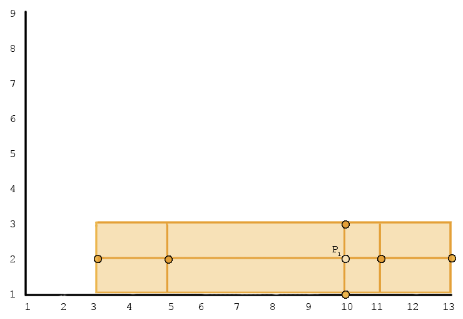
When working in a finite element setting we will need elements to do Gaussian quadrature over. When using T-splines, these can *not* be chosen as the T-mesh faces. Indeed from definition 3.2.1, there does not exist anything such as a T-mesh face. This is due to the fact that the T-spline technology allows for I-joints, L-joints and even zero-joints. While we have not explored I-joints and zero-joints in this thesis it is noteworthy that they might provide interesting refinement properties. L-joints, however *will* be existing in our T-spline solvers. It is obvious that from an implementation point of view, it would be horrible to do Gaussian quadrature over such faces (as the shaded face in figure 3.3b), but it would also have very bad approximation properties due to the fact that the blending functions are not guaranteed to be C^∞ over these. What we will instead do, is define the elements as follows

Definition 3.5.1 A T-spline element is a rectangular face over which every blending function is C^∞

To see this problem in detail, we take a close look at the support of the basis function centered at P_1 in figure 3.4. We have already seen that the index knot vectors are $I = [3, 5, 10, 11, 13]$ and $J = [1, 2, 3]$. Since it is a B-spline at all these points, the T-spline will have reduced continuity over these values. We see from figure 3.5a the support of the basis function R_1 as the shaded area. What is the key here, is to observe that the continuity reduction lines splits the face $[11, 13] \times [1, 6]$ in three. Across every line of



(a) The support of the function R_1 centered at P_1



(b) All continuity reduction lines corresponding to P_1

Figure 3.5: Continuity reduction

figure 3.5b we have reduced continuity and they will potentially split faces in two. For the particular case of R_2, there is only two lines, namely the line $j = 2$ and $j = 3$ that are splitting anything. All the other lines are lining up with existing lines of the T-mesh. In the case of B-splines, every single continuity reduction line will be aligning with existing B-mesh lines, and thus it is not necessary to do the same steps for such bases.

For any T-spline we will however need to do the element extraction before any analysis can be performed. This is done by drawing the continuity reduction lines corresponding to each and every blending function. After this has been done, we are to extract the elements from the net of lines. Visually, this is very easy, it's just drawing each line and the elements appear as the obvious rectangles. It is slightly tedious to do this by computational power, and details are omitted here. To continue with the same example, the elements corresponding to the mesh drawn in figure 3.4 is shown in figure 3.6. We here see that the elements almost span out every full row and column. This is very special for this particular T-mesh since the refinements are so much spread out over the entire domain. If we were to create a more natural T-mesh, this would be subject to local refinement and you would have a very course mesh in some areas, and a very fine mesh in other ares. The fine mesh would generate reduced continuity lines for a very limited range, and these would not span far into the course mesh (see chapter 6 for examples). In this particular mesh, we have scattered areas of fine mesh which is a bad thing to have when it comes to the number of elements generated.

There are issues one should pay extra attention to when it comes to handling these elements. Remember the T-mesh is generating the blending functions and is in many ways defining our T-spline. The refining process which is taking place in the index domain is using the T-mesh as it is defined there. However the elements seem to be a bit detached from this. Consider for example the element $(i, j) \in [6, 7] \times [6, 7]$ in figure 3.6. This element is completely contained within the T-mesh "face" $(i, j) \in [3, 9] \times [5, 9]$. Consider if we had an iterative scheme which continuously refined the elements, and had marked the element $[5, 6] \times [5, 6]$ for refinement. Since it is impossible to insert blending

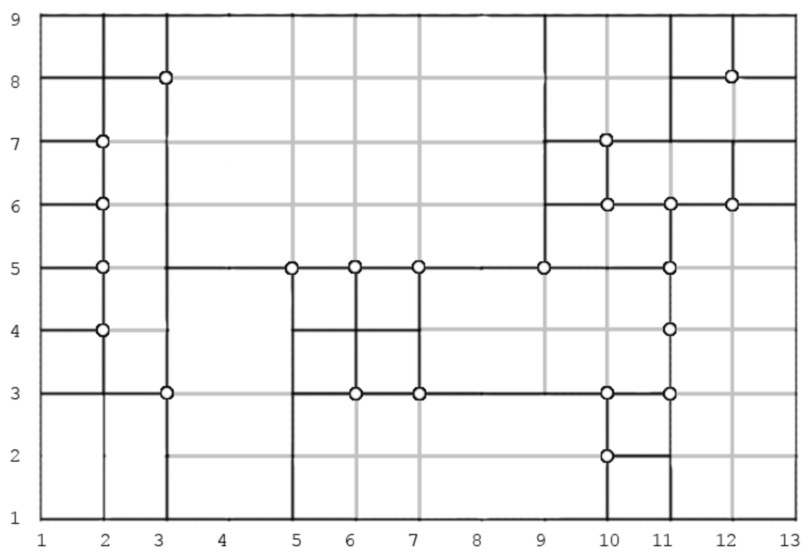


Figure 3.6: The elements corresponding to the example T-mesh. Continuity reduction lines are drawn in gray, and the points which generate these lines are marked by circles

functions alone by the refining algorithm, we would have to search for the nearest edge in the T-mesh which we could split, which in turn would split the element in two. There is a non-trivial interaction between the elements and the T-mesh which is taking place in the refining process. For the usual FEM assembly algorithm (described later in algorithm 2) it is however enough to extract the elements from the T-mesh and work on with these to assemble the stiffness matrix.

Chapter 4

NURBS as a FEM basis

This chapter introduces splines as a basis for Finite Element methods (FEM). We will assume that the reader is already familiar with classical FEM using more established basis functions such as linear, bilinear, Gauss-Legendre polynomials etc. We will briefly introduce the notation for the FEM problems at hand and point out the differences which do arise when using splines as basis functions as opposed to classical choices. While this chapter will at all points talk about NURBS, it is not hard to generalize this to other splines such as the T-splines.

4.1 Isogeometric

There are many existing ways of dealing with complex geometry in classical FEM. Among these are the Gordon-Hall algorithm [12] in which the FEM-solver is using a small number of high-order elements, or triangulation which is often used with linear elements. These two approaches are very descriptive of how the choice of geometric representation is picked. With the linear basis functions, we often choose a linear geometry, which is made up of triangles. With the Gordon-Hall algorithm we are typically using high-order basis functions, and the same basis functions for describing the geometry. In many ways it is the choice of mathematical basis functions which are dictating how we are to represent the geometry. The problem is imposing the restrictions on the geometry. In an *isogeometric* context this relationship is turned the other way around. In this case it is the geometry which will impose the restrictions on the FEM basis-functions. We will choose a superior way of representing the geometry, and armed with this model, we will use the same basis functions to create a Galerkin projection of the solution.

In the introduction we mentioned that the choice of splines as basis functions would ease the transition from any CAD-model to a suitable FEM-model. While this is the case it is also believed that the spline basis functions are numerically superior when it comes to solving the problem on challenging geometrical constructs. This is much due to the fact that the error in the description of the geometry will be much more precise. In fact, it is often considered that the spline model coming from a CAD program is defined to be the exact geometry and in those cases you will have zero error contribution from geometrical factors.

4.2 Nodal values

In classical FEM it is very popular to use nodal basis functions. This means that the domain is divided into nodal points and elements which connect them. For each node p_i there exists a basis function v_i which is nonzero (usually one) at the node itself, and zero at every other node. This gives a very intuitive interpretation of the solution since it is easy to visualize this as the surface going through each point with the given nodal solution value. In between the nodes however, there are different interpretations. Here, you would typically need to explicitly sample the function, which would require more or less computation depending on your choice of basis functions. For all nodal basis functions however it is fairly straightforward to interpret or visualize the solution. This is however no longer the case when dealing with spline basis functions. The solution vector $\boldsymbol{u}$ will contain the contribution of the different basis functions, but it is not readily apparent what the values of the solution is at any given points. To prevent any misunderstandings or false analogs to other classical FEM cases, we will restrain ourself from using the word "node", and will instead always talk about "index point", "knot point" or "control point" depending on whether the point in question is in the index-, parametric- or physical domain.

4.3 Elements

The choice of elements is not always as apparent as one might hope. When it comes to B-splines and NURBS, we will define each tensor product knot span as an element. This has some consequences such as the basis functions having support over multiple elements. There is a common misunderstanding that this will cause the stiffness matrix to no longer being sparse. This is however not the case as the basis functions still have *local* support. It is just the term *local* which will need to be redefined. From chapter 2 it was shown that the support of any B-spline basis function was exactly over $p+2$ knots in the knot vector. This gives a maximum (depending on the existence of multiple knots) of $p+1$ knot spans or elements in each parametric direction. It is to be understood that p is typically much smaller than n, so the support is truly local.

With T-splines the choice of elements is even less obvious. They will however be defined as in the previous chapter, as the rectangles in which all blending functions are C^∞. This information will have to be extracted prior to doing any numerical integration during the stiffness matrix assembly.

It is also noteworthy to realize that when using spline basis functions, the continuity will be very different from classical FEM methods. With splines, we will have a much greater control of the continuity across element edges. Using classical finite elements, this would usually be C^0 over each edge. While it was possible to add additional requirements on the edges by hand to overcome this, you will typically get this "for free" when using splines. The exact continuity is dependent on the spline in question, but with simple

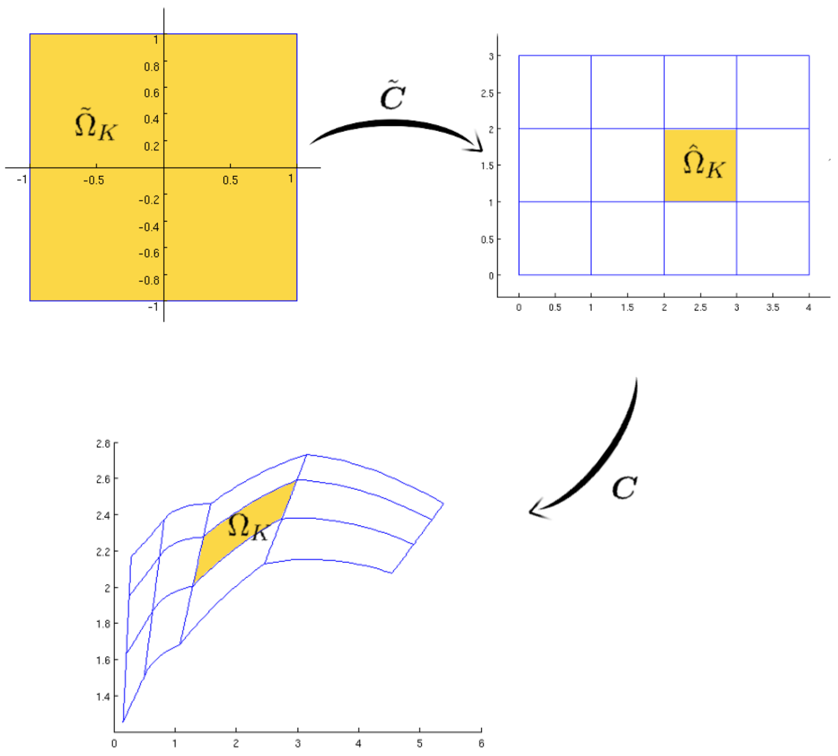


Figure 4.1: The unit square $\tilde{\Omega}$, parametric space $\hat{\Omega}$ and physical space Ω

knots, you will have C^{p-1} continuity across element boundaries. This has impacts on things such as the a posteriori error estimates. If quadratic splines with no duplicate knots are used, you are guaranteed to get a continuous flux of the solution. This will again mean that you can skip some of the error contributions arising from discontinuities. See the next chapter on a posteriori error estimation for details.

4.4 Numerical integration

During the assembly of the stiffness matrix, we will typically need to do numerical integration. We will in this thesis use Gaussian quadrature as our choice of numerical integration. To do this, we will however need to map our functions over to a unit square $(\tilde{\xi}, \tilde{\eta}) \in [-1, 1] \times [-1, 1]$. While this is common for most finite element methods as well, we have the additional mapping from the parametric space to the physical space. That is we are dealing with *two* mappings. One which is an affine mapping $\tilde{C}$ from the unit square $\tilde{\Omega}_K$ to the parametric element $\hat{\Omega}_K$, and one which is the (rational) polynomial mapping $\boldsymbol{C}$ from the parametric element to the physical element Ω_K. See figure 4.1. This relationship is quite important to realize as we will be working in all spaces at the same time. The numerical integration has to take place in the unit square, while all basis functions are defined over the parametric domain and the differential equation is formulated in the physical space.

The fact that the functions themselves are defined in the parametric space while the differentials are defined in the physical space makes it necessary to reformulate the equations. Let the Jacobian J and its inverse be defined as

$$J = \begin{bmatrix} x_\xi & x_\eta \\ y_\xi & y_\eta \end{bmatrix}, \quad J^{-1} = \begin{bmatrix} \xi_x & \xi_y \\ \eta_x & \eta_y \end{bmatrix} \tag{4.1}$$

Note that we do not have any of the derivatives in J^{-1} available from a computational point of view. The mapping $\boldsymbol{C}(\xi,\eta) = \begin{bmatrix} x(\xi,\eta) \\ y(\xi,\eta) \end{bmatrix}$ is in general a non-degenerate polynomial, which can been shown by any elementary algebra book [11] to not have an available inverse function for any polynomial of degree greater than four by. Since $\boldsymbol{C}$ is not (exactly) invertible it is therefore necessary to derive some identities at this point. From the fact that we can also calculate the inverse of the Jacobian by basic linear algebra we arrive at the following equations for the derivatives

$$J^{-1} = \frac{1}{x_\xi y_\eta - x_\eta y_\xi} \begin{bmatrix} y_\eta & -x_\eta \\ -y_\xi & x_\xi \end{bmatrix} = \begin{bmatrix} \xi_x & \xi_y \\ \eta_x & \eta_y \end{bmatrix} \tag{4.2}$$

Let us also define the gradient in the different spaces as

$$\boldsymbol{\nabla} = \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{bmatrix}, \quad \hat{\boldsymbol{\nabla}} = \begin{bmatrix} \frac{\partial}{\partial \xi} \\ \frac{\partial}{\partial \eta} \end{bmatrix}. \tag{4.3}$$

It can then be shown by the chain rule that the relationship between these are

$$\boldsymbol{\nabla} = \begin{bmatrix} \xi_x & \eta_x \\ \xi_y & \eta_y \end{bmatrix} \hat{\boldsymbol{\nabla}} = \boldsymbol{G}_\nabla \hat{\boldsymbol{\nabla}} \tag{4.4}$$

The values of $\boldsymbol{G}_\nabla$ can now evaluated using the identities (4.2). We will typically need to evaluate the gradient of basis functions during the creation of the stiffness matrix in i.e. the Laplace equation (see next chapter). However we will also sometimes require the evaluation of ∇^2 of the basis functions. In this case, we will use (4.4) twice

$$\nabla^2 = \boldsymbol{\nabla}^T(\boldsymbol{G}_\nabla \hat{\boldsymbol{\nabla}}) = (\boldsymbol{G}_\nabla \hat{\boldsymbol{\nabla}})^T \boldsymbol{G}_\nabla \hat{\boldsymbol{\nabla}} \tag{4.5}$$

Since $\hat{\boldsymbol{\nabla}}$ is only derivatives with respect to ξ and η and all functions in $\boldsymbol{G}_\nabla$ are available, this can be expanded by continued use of the chain rule. The result is far from pretty and details are omitted for the sake of brevity. The evaluation of ∇^2 is used in the a posteriori error estimates (see the next chapter).

Chapter 5

Example problems

To illustrate the numerical properties of the isogeometric finite element method, we will implement this to solve it on two popular [17] benchmark examples. The first of these is the stationary heat equation, or Laplace equation over an L-shaped domain. One of the properties of this particular problem is that it has a singularity at the origin, which will destroy the convergence of uniform refinement. It is therefore needed to provide some form of local refinement, and T-splines will in this case prove superior. The second problem is one from linear elasticity. It descirbes an infinite plate with a circular hole in it, and a constant applied stress at the each end. Both problems have exact solutions available, and due to their widespread use, should provide the reader with some familiar ground when it comes to comparing the performance of the techniques applied here.

5.1 Laplace equation

The problem consist of solving the stationary heat equation $\nabla^2 u = 0$ on a L-shaped domain $\Omega = [-1,1]^2 \setminus [0,1]^2$ with appropriate boundary conditions.

$$\begin{array}{lll} \nabla^2 u & = 0 & \text{in} \quad \Omega \\ u & = 0 & \text{on} \quad \Gamma_D \\ \frac{\partial u}{\partial \boldsymbol{n}} & = g & \text{on} \quad \Gamma_N \end{array} , \tag{5.1}$$

with $g(x, y)$ given by the exact solution at the Neumann edge and $\boldsymbol{n}$ being an outward unit normal. It can be shown that

$$f(r,\theta) = r^{2/3} \sin\left(\frac{2\theta + \pi}{3}\right) \tag{5.2}$$

is a solution to the Laplace equation $\nabla^2 u = 0$, and this is what we will be using as our exact comparison solution. The generation of g is straightforward from f but is not given as a simple expression and the details are omitted here. The homogeneous Dirichlet boundary is given as $y = 0, x \in [0,1]$ and $x = 0, y \in [0,1]$, while all other edges are given with Neumann conditions (see figure 5.1). Note that the exact solution, which

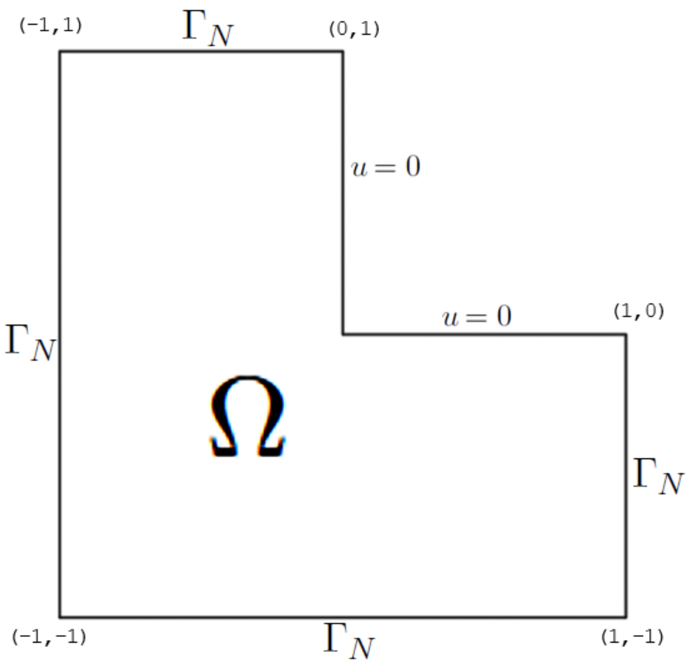


Figure 5.1: The domain Ω along with the boundary conditions

is pictured in figure 5.2 shows the singularity at the origin. The function has a sharp edge at that point, and the derivative is not defined here.

5.2 Variational formulation

We are going to solve (5.1) using a finite element method framework. To do this we will need to find the weak form of the problem stated above. This is done by integrating each side of the equation by a spline test function R. The equation then becomes

$$\int_\Omega \nabla^2 u \cdot R \, dA = 0, \tag{5.3}$$

which after integration by parts and rearranging the terms becomes

$$\begin{aligned} \int_\Omega \boldsymbol{\nabla} u \cdot \boldsymbol{\nabla} R \, dA &= \int_\Gamma \frac{\partial u}{\partial n} R \, dS & (5.4) \\ a(u, R) &= l(R) & (5.5) \end{aligned}$$

Where a and l is the usual bilinear and linear functions introduced in any elementary book on FEM, i.e. [5]. It should come as no surprise that we are going to use splines as

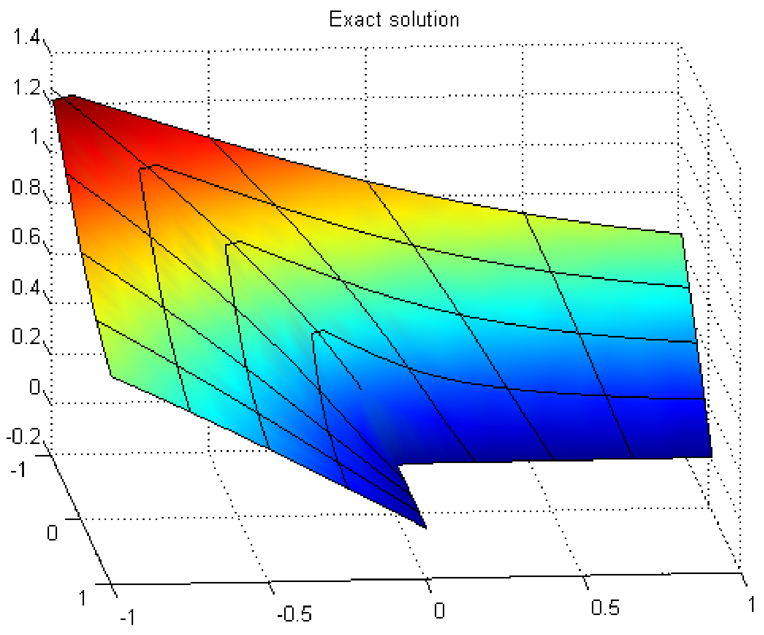


Figure 5.2: The exact solution of (5.1)

our test functions, hence the name R as the test function, opposed to the more generally used v as seen in classical finite element literature. We are going to form a Galerkin projection of the exact solution into a subspace of $\mathcal{S}$. This subspace will be built up of all spline functions of a specific type, i.e. NURBS, B-splines or T-splines which will map the surface to the physical domain Ω. We will then find a subspace $\mathcal{S}_n$ which only span a finite number of spline basis functions. That is

$$\mathcal{S}_n = span\left\{ R_i \in \mathcal{S} \,\middle|\, \boldsymbol{C}(\xi,\eta) = \sum_{i=1}^{n} R_i \boldsymbol{B}_i = \Omega \right\}. \tag{5.6}$$

Thanks to the precise formulation of knot insertion, it is possible to iteratively increase the size of $\mathcal{S}$ while still preserving the mapping $\boldsymbol{C}$ unchanged.

While searching for a solution in the Galerkin subspace, it is understood that the solution can be written as

$$u_h = \sum_{i=1}^{n} u_h^i R_i. \tag{5.7}$$

(5.7) inserted in (5.5), along with exploiting the bilinearity of a gives

$$\sum_{i=1}^{n} a(R_i, R_j) u_h^i = l(R_j), \quad \forall R_j \in \mathcal{S}_n \tag{5.8}$$

5.3 Assembly

(5.8) is a linear system of n equations which we will solve by creating the matrix $\boldsymbol{A} = [a(R_i, R_j)]$ and the vector $b = [l(R_j)]$ such that

$$\boldsymbol{Au} = \boldsymbol{b} \tag{5.9}$$

The construction of the stiffness matrix $\boldsymbol{A}$ is often referred to as "assembling" due to its method of construction. While the formulation is defined as an integral over pairs of basis functions, it is more convenient to loop over each element instead of looping over all basis functions at the outer loop. The algorithm of constructing $\boldsymbol{A}$ is described in algorithm 2.

Algorithm 2 Assembly of $\boldsymbol{A}$

```
1: for all elements $e$ do
2:   for all nonzero basis functions $R_i$ do
3:     for all other nonzero basis functions $R_j$ do
4:       $I^* \leftarrow 0$
5:       for all Gaussian points $g$ do
6:         $I^* \leftarrow I^* + a_g(R_i, R_j)$
7:       end for
8:       $A_{ij} = A_{ij} + I^*$
9:     end for
10:  end for
11: end for
```

Line 5-7 is the Gaussian approximation $I^* = \sum_g a_g(R_i, R_j)$ of the exact integral $I = a(R_i, R_j)$. It is a well known fact that Gaussian quadrature has superior qualities when approximating polynomials [6]. B-splines are regular polynomials over each element, but even in the case of *rational* polynomials such as NURBS, experiments have shown that Gaussian quadrature gives a very good approximation [8]. There have been some recent research in the choice of quadrature when using NURBS-based isogeometric analysis. Hughes et al. (2008) [16] devised a technique which utilized roughly half the number of quadrature points as you had degrees of freedom. While we take note that there exists alternative choices of quadrature, we have chosen to go with the regular tensor product Gaussian quadrature, mainly because of it's popularity and familiarity.

There are several issues which we should take note of when implementing this algorithm. First of all, the extraction of the elements themselves are not trivial when it comes to T-splines. In line 2-3 of algorithm 2 we are looping over each nonzero basis function. Usually this is implemented as a list, i.e. to each element e, there is a list L_e over every nonzero basis function at that element. This is also necessary for T-splines, but with B-splines and NURBS it is not. Thanks to the well defined support of the splines over the index domain we can easily extract the nonzero basis functions by arithmetic means, and will not need lookup tables. Also note that the Gaussian integral itself is subject to two mappings which will influence its computation.

With a from (5.5) we have that the exact integral is

$$a(R_i, R_j) = \int_\Omega (\boldsymbol{\nabla} R_i)^T (\boldsymbol{\nabla} R_j)\, dxdy \tag{5.10}$$

with $\boldsymbol{\nabla}$ being taken with respect to x and y. R_i on the other hand, is expressed in ξ and η. Using (4.4) we have that

$$\begin{aligned}
a(R_i, R_j) &= \int_\Omega (\boldsymbol{G}_\nabla \hat{\boldsymbol{\nabla}} R_i)^T (\boldsymbol{G}_\nabla \hat{\boldsymbol{\nabla}} R_j)\, dxdy. && (5.11)\\
&= \int_{\hat{\Omega}} (\hat{\boldsymbol{\nabla}} R_i)^T \boldsymbol{G}_\nabla^T \boldsymbol{G}_\nabla (\hat{\boldsymbol{\nabla}} R_j) \det(J)\, d\xi d\eta && (5.12)\\
&= \int_{\tilde{\Omega}} (\hat{\boldsymbol{\nabla}} R_i)^T \boldsymbol{G}_\nabla^T \boldsymbol{G}_\nabla (\hat{\boldsymbol{\nabla}} R_j) \det(J) \det(\tilde{J})\, d\tilde{\xi} d\tilde{\eta} && (5.13)\\
&\approx \sum_{\alpha,\beta} (\hat{\boldsymbol{\nabla}} R_i(\xi_\alpha, \eta_\beta))^T (\hat{\boldsymbol{\nabla}} R_i)^T \boldsymbol{G}_\nabla^T \boldsymbol{G}_\nabla (\hat{\boldsymbol{\nabla}} R_j(\xi_\alpha, \eta_\beta)) \det(J) \det(\tilde{J}) \rho_\alpha \rho_\beta
\end{aligned}$$

where we in (5.12) have mapped the area of integration to the parametric space (ξ, η), and in (5.13) have mapped it yet another time, to the unit square. In the last line, we have used Gaussian quadrature to evaluate the integral with ρ being the Gaussian weights associated to the evaluation points (ξ_α, η_β). Moreover, the mapping $\tilde{\boldsymbol{C}}$ from the Gaussian domain to one element in the parametric domain is an affine mapping thus reduces the determinant of the Jacobian $\tilde{J}$, corresponding to this mapping, to a constant. This constant will be equal to the relative difference between the area of the parametric element and the area of the unit square.

The construction of the load vector $\boldsymbol{b}$ follows the exact same progress with looping over each element and successively adding up each contribution to the load vector from each nonzero basis function by evaluation of a Gaussian quadrature.

5.4 Linear elasticity

This problem features an infinite plate with a circular hole in it. A uniform stress is applied in x-direction away from the hole. This problem has an exact solution which can be found in Zienkiewicz [23]. Due to the symmetry, it is sufficient to consider only the first quadrant of the plate. The governing equations are

$$\begin{aligned}
\boldsymbol{\nabla} \cdot \boldsymbol{\sigma}(\boldsymbol{u}) &= 0 \quad && \text{in } \Omega\\
\boldsymbol{u} &= g_D && \text{on } \partial\Gamma_D\\
\boldsymbol{\sigma} \cdot \boldsymbol{n} &= g_N && \text{on } \partial\Gamma_N,
\end{aligned} \tag{5.14}$$

with the problem geometry pictured in figure 5.3. There is homogeneous Neumann conditions on the inner circle, symmetry boundary conditions on the south and west edge, and finally exact Neumann boundary conditions given by the exact stress on the east and north side of the domain.

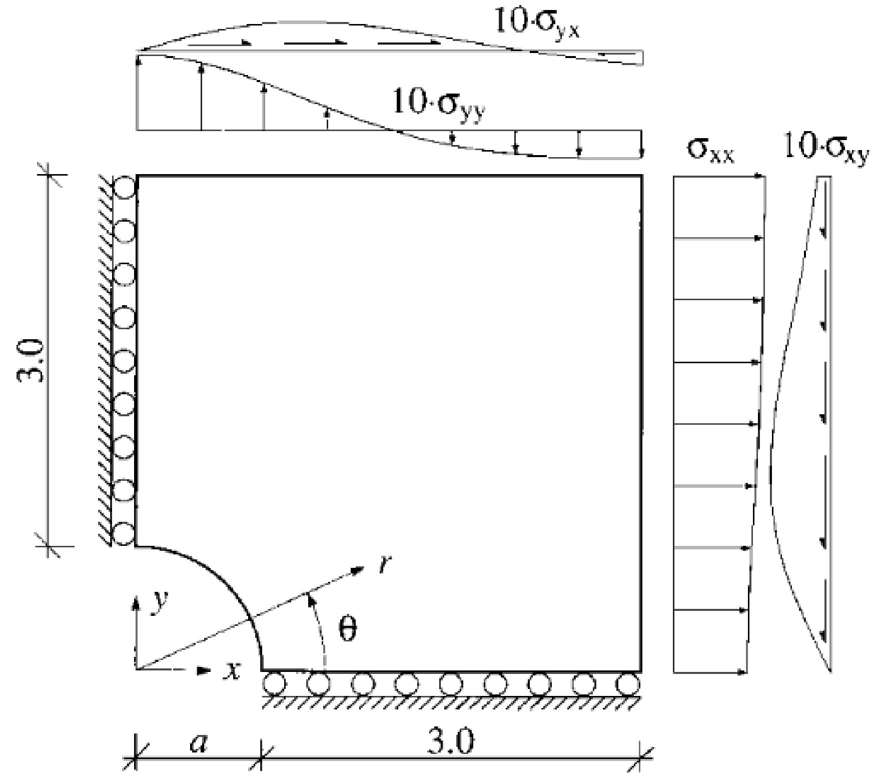
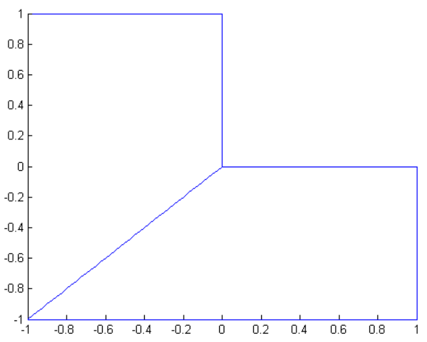


Figure 5.3: The linear elasticity setup

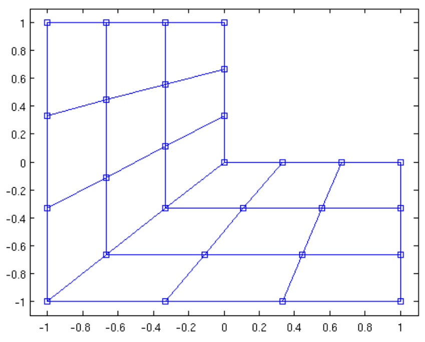
While the problem description of (5.14) takes on a seemingly different form than that of the Laplace equation (5.1) the assembly of the stiffness matrix is almost identical. It uses the same loop as the one described in algorithm 2. The main difference is that the unknown displacement field $\boldsymbol{u}$ is now a vector which will slightly alter the process. This is handled by introducing a basis function for each dimension for each knot point. That is

$$\boldsymbol{u_h} = \sum_{i=1}^{n} \sum_{d=1}^{2} u_h^{i,d} R_i(\xi, \eta) \boldsymbol{e}_d, \tag{5.15}$$

where $\boldsymbol{e_1} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\boldsymbol{e_2} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. We are then looping over twice as many basis functions, since each basis function will consist of the regular spline basis function R, along with the spacial direction which it is active in, i.e. $\boldsymbol{e}_d$. Other than this, the assembly of $\boldsymbol{A}$, the solution of the linear system of equations and the grid refinements will all be completely analogue to what we saw when we solved the Laplace equation.



(a) The coarsest element mesh used for solving the Laplace equation. The lines illustrating the knot spans



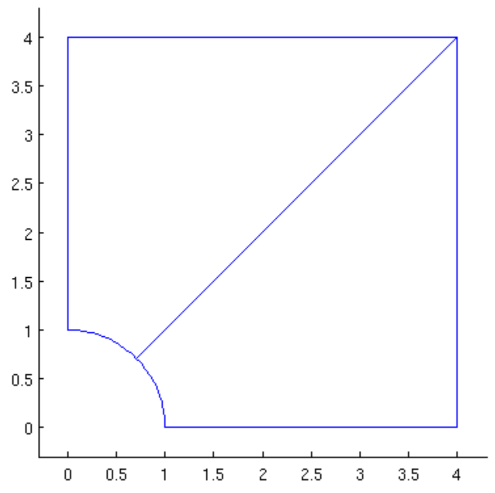
(b) The control mesh corresponding to the parametrization. The squares are the control points

Figure 5.4: L-shape parametrization

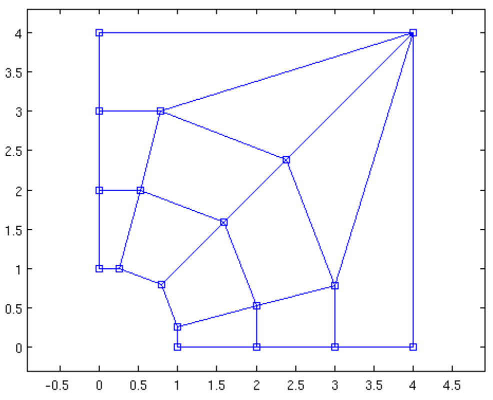
5.5 Geometry construction

The geometry of the L-shaped problem is not particularly suited just for splines, since it is easily described by other, simpler, geometric constructs. Nevertheless, it is such a popular benchmark example that it was decided to be included. Also it will allow for the local refinement by T-splines to shine and come to their best use due to the singularity at the origin. The geometry was constructed in the following way. It was designed as a cubic NURBS with $\Xi = [0, 0, 0, 0, 1, 1, 1, 2, 2, 2, 2]$ and $\mathcal{H} = [0, 0, 0, 0, 1, 1, 1, 1]$, that is only the endpoints are included in the η-direction, while the ξ-parametrization has an interpolating point halfway through the parameter space. This allows us to create the sharp edge at the origin. It will split the domain into two elements by the diagonal axis as indicated in figure 5.4a. The control points were uniformly positioned within the domain as indicated by figure 5.4b.

The circle with a hole was constructed in a slightly different manner. A second order circle was created at the origin from an available template. Of this, only the first quadrant was kept, after which order elevation was performed, along with a knot insertion at the middle of the domain. The third order half-circle was linearly extruded out to a radius of 4. Finally the three top right control points were placed on top of each other to make the sharp edge. Multiple control points are causing the derivatives to vanish, much in the same manner as duplicate knots are doing. This allowed us to create an interpolating sharp edge at the top right corner. The elements and control mesh corresponding to the final knot vectors $\Xi = [0, 0, 0, 0, 1, 2, 2, 2, 2]$ and $\mathcal{H} = [0, 0, 0, 0, 1, 1, 1, 1]$ is displayed in figure 5.5 Note that this parametrization has some very attractive properties. If we take a look at the knot vectors, we see that there is not a single multiple knot, save the start- and endpoint. This means that all basis functions have high continuity at the interior of the domain. In fact, since we have a cubic spline with a single knot, the most reduced continuity is at the line $\xi = 1$ which will have $p - 1 = 2$ continuous derivatives. At the



(a) The coarsest element mesh used for solving the Linear elasticity equation



(b) The control mesh corresponding to the parametrization

Figure 5.5: Circle with a hole parametrization

multiple control point at $(x, y) = (4, 4)$, the continuity is further reduced, but this is only at this single point. We will never place any Gaussian quadrature points here, and will thus never evaluate the functions at all at this point so it is fine. The beauty of having high continuity across the element boundaries is that we can now have a continuously defined stress. This will in turn mean that we don't have to pay any special attention to the jump error which you traditionally get from using C^0 element boundaries in a classical finite element solver. That is not to say that this particular parametrization comes without its drawbacks. With so many control points in the top right corner, the element refinement lines tend to be drawn towards this point. The "uniform" refinement scheme is thus no longer so uniform when viewed from the physical domain (even if it is completely uniform in the parametric domain). It is however a choice we make, and we will prefer increased continuity over an otherwise uniform parametrization. As we shall see in the next chapter, the T-splines are proving to have an interesting effect on this.

5.6 A posteriori error estimation

In order to achieve local refinement we will need some form of feedback on where the error is larger. This is the fundamental starting point for local refinement. One way of doing this is trying to evaluate the error over each element separately in some norm. Usually this is chosen to be the energy norm. This is the natural choice of norm since the FEM is minimizing the distance between the true solution u and the discrete solution u_h in this norm. Since we actually have the exact solution available, the exact error can be evaluated (up to Gaussian quadrature approximation) and this can be used for refinement purposes. We can evaluate the exact error by

$$|||e|||^2 \equiv \|e\|_E^2 = a(e,e) = a(u-u_h, u-u_h) \tag{5.16}$$

or by restricting this to each element Ω_k by

$$\alpha_k = |||e|||_{\Omega_k}^2 = a_{\Omega_k}(e,e)^2 = \int_{\Omega_k} (\boldsymbol{\nabla}(u-u_h))^T \boldsymbol{\nabla}(u-u_h) d\Omega. \tag{5.17}$$

Note that we have dubbed the element-error α_k and not η_k, which is frequently seen in the literature [2] to distinguish it from the extensive use of η as a parameter-space axis in this thesis. By doing this evaluation, we will have an exact value α_k of the error for each element, and we can proceed with refining the elements which contribute the most to the global error $|||e|||$. While this strategy is perfectly valid if we have the exact solution available, it is unlikely that this will be the case in a practical engineering problem. If the exact solution is not available, then we will instead try and approximate this as best we can. Consider the Laplace equation formulated for the L-shaped domain described previously. The derivation for the hole-problem is completely analogous. If we have an approximate solution u_h available, then we can modify our original equation by

$$\begin{aligned} \nabla^2 u &= f \\ \nabla^2 u - \nabla^2 u_h &= f - \nabla^2 u_h \\ \nabla^2 e &= f - \nabla^2 u_h. \end{aligned} \tag{5.18}$$

This is an exact formulation of the error $e = u - u_h$. It is in fact the exact same equation as our original Laplace equation, with the only exception that we are approximating the unknown e instead of u, and the right hand side has been changed from the known function f to the known function $f - \nabla^2 u_h$. It is obvious that we could use the exact same framework to solve this equation, as we did to solve our original problem. This is however not a good thing to do due to several factors. First of all it is well known from finite element analysis that the Galerkin projection is minimizing the error (measured in the energy norm). This is known as the Galerkin orthogonality property

$$a(e,v) = 0, \quad \forall v \in \mathcal{S}_n. \tag{5.19}$$

If we project the error into the finite space $\mathcal{S}_n$ then the error would be identical to zero. Secondly, there is the fact that the evaluation of the error itself would be just as costly as the original approximation of the solution. In any practical applications this is not acceptable. Dörfel et al. [10] suggests projecting this error into another subspace $\mathcal{W}$ which is spanned by the so-called bubble functions over the element interiors. This was tried implemented, but gave less than satisfactory results. We will instead rely on the theory of a posteriori error estimation as presented by Ainsworth and Oden [1].

To derive the error approximation we note that

$$a(e,v) = a(u,v) - a(u_h,v) = l(v) - a(u_h,v), \quad \forall v \in \mathcal{S}, \tag{5.20}$$

which is basically the same as (5.18) written in weak form. Since we are interested in an error contribution from each separate element, we will first need to decompose the error into the contribution from each element. Let $\mathcal{P}$ be the partition of the domain Ω into elements. Then

$$\begin{aligned} a(e,v) &= l(v) - a(u_h,v) \\ &= \sum_{K\in\mathcal{P}} \int_K fvdA + \int_{\partial K\cap\Gamma_N} gvdA - \int_K \boldsymbol{\nabla} u \boldsymbol{\nabla} vdA \end{aligned} \tag{5.21}$$

where ∂K is the border of element K and Γ_N is the outer Neumann boundary. The above statement is true not only for all test functions in our solution subspace, but for all $v \in \mathcal{S}$. Integration by parts over each element gives

$$a(e,v) = \sum_{K\in\mathcal{P}} \int_K rvdA + \int_{\partial K\cap\Gamma_N} RvdA - \int_{\partial K\setminus\Gamma_N} \frac{\partial u}{\partial \boldsymbol{n}} dS \tag{5.22}$$

where we have defined the interior residual r and boundary residual R by

$$r = f + \nabla^2 u \tag{5.23}$$

$$R = g - \frac{\partial u}{\partial \boldsymbol{n}}. \tag{5.24}$$

The last term in (5.22) is taken over all inter-element borders. Ainsworth [1] is approximating these values by taking the mean of $\frac{\partial u}{\partial \boldsymbol{n}}$ on each side of the element border. This is done since in regular finite element methods the basis functions will be C^0 across element boundaries which results in a jump discontinuity in the approximation of the flux. This is not the case with splines as they have increased continuity across element boundaries. The hole geometry was created with this in mind as well. The final term then vanishes as each inter-element line integral will cancel each other. We are left with

$$a(e,v) = \sum_{K\in\mathcal{P}} \int_K rvdA + \int_{\partial K\cap\Gamma_N} RvdA \tag{5.25}$$

By continued use of the Cauchy-Schwartz inequality and exchanging the general function v with the error, we arrive at the result

$$|||e|||^2 \leq C \left\{ \sum_{K\in\mathcal{P}} h_K^2 \|r\|_{L^2(K)}^2 + \sum_{\gamma\in\Gamma_N} h_K \|R\|_{L^2(\gamma)}^2 \right\}, \tag{5.26}$$

where h_K is the diameter of element K. In the implementation, this was chosen to be the largest diameter. For details on the complete derivation, see Ainsworth [1]. Note that the last term in (5.26) will only be nonzero for the elements which border a Neumann boundary condition edge.

Note that this error estimate is not especially good for evaluating the actual numerical value of the error in question. The constant C in (5.26) is not known, and the bond of the inequalities are not especially tight. However this error estimate will give a very good indication on what the *relative* error between the elements will be. This is sufficient for our purposes as it will enable us to tag the elements which are contributing the most to the error, and we can thus proceed with refining these elements. In an industrial implementation this error estimate will allow for solution schemes where you could request the degrees of freedom you would want, and the algorithm will proceed with finding the optimal discretization of this. However, it will not allow the user to input a maximal numerical error value, as we have no information on the constant C.

Chapter 6

Numerical Results

In this chapter we will present the results on how the algorithms described perform on the actual problems. It is in general a bad idea to compare different techniques with respect to computer running time. This is very dependent on the implementation itself, and also very dependent on the choice of programming language. What we will instead do is compare the convergence rate as a function of the degrees of freedom. This comparison has its drawbacks as well since you would probably be able to handle more degrees of freedom for the same amount of computational power with a simpler method. Nevertheless, we are going to present the rates in which the methods converge with emphasis on how the T-splines perform against tensor-product NURBS structures on the same problem.

6.1 L-shaped Laplace equation

When refining the mesh, several approaches were tried. First and most obvious was the uniform refinement. This was done by recursively halving the interval between consecutive knots, it is displayed in figure 6.1.

Needless to say, the convergence of the uniform refinement was less than satisfactory. With the knowledge of there being a singularity at the origin, it is obvious that we should have some form of local refinement around this point. Another strategy was devised with the refinement only taking place around this very point. What we did was to start with a slightly refined uniform mesh (given in figure 6.1) and after this halve all elements closest to the singularity. The result was the mesh given in figure 6.2. It had severely improved convergence, but as the degrees of freedom increased, the error in the non-refined pieces of the mesh was becoming dominant and the convergence declined.

The uniform refinement did, not unexpected, perform poorly due to the presence of the singularity at the origin. With cubic NURBS we still only achieved a convergence of order 1/2. The rule of thumb refinement scheme proved vastly superior. It is expected for polynomial elements to have a convergence rate of $\mathcal{O}(h^{p+1})$. With two spacial dimensions, the element size goes as $1/h = \mathcal{O}(\sqrt{N})$ which means that with cubic elements we are expecting second order convergence as a function of the total number of basis functions

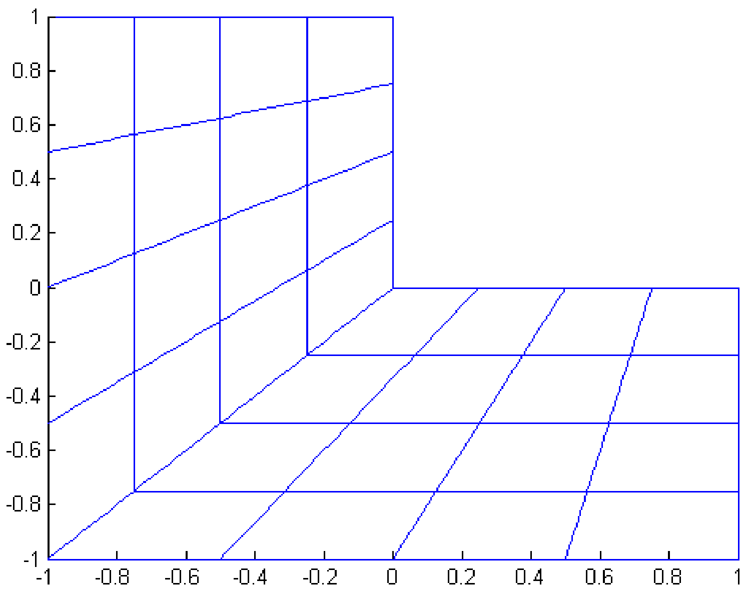


Figure 6.1: Uniform mesh after two refinements

N. This is clearly seen in figure 6.3. The scheme does however get to a point where the error in the non-refined parts of the mesh becomes dominant. This scheme does never refine anything but the innermost element.

The story becomes completely different when introducing the a posteriori error estimate. This does not suffer the same effect of flattening out as the rule-of-thumb refinement strategy does. The scheme was to first find an approximate solution. After this we evaluated an approximate error at every element. The top 1 percent of the elements with the largest error was then refined. The reason for choosing such a small number, is that up until a very late stage of the solution process, the dominant error was only at the innermost elements and refinements in other places would not yield a significantly better solution. We see that the local error estimator successfully detects the singularity at the origin and refines around this. It is also detecting the elements which are secondary most prone to high error. These are the ones along the diagonal line across the L. However with the NURBS tensor product structure, there is a lot of unnecessary refining taking place. The elements at the end are only receiving refinement from the more square center nodes which are placing a line through the entire domain. This results in the characteristic long elements at the edge which is seen in figure 6.4b. These elements themselves are never going to be subject to refinement (which could make them more square) since they are too far from the singularity to have any significant error, and they do also have too small areas to compete with any of the other elements.

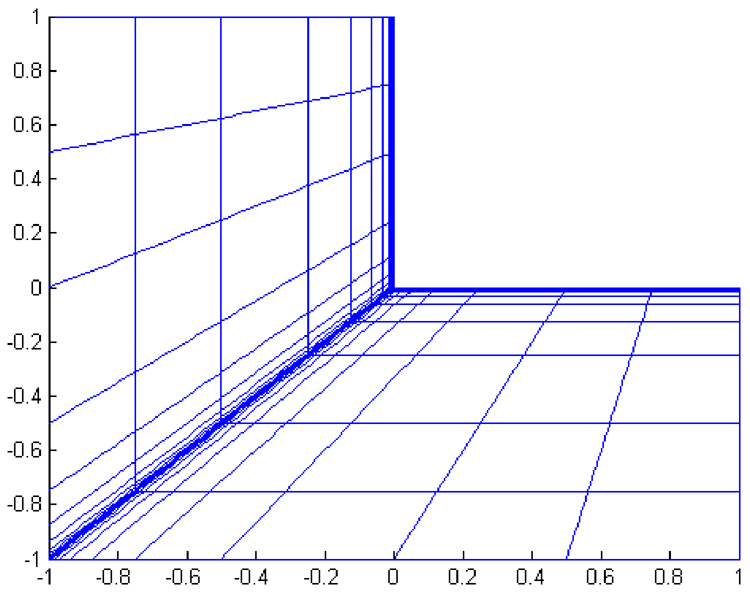


Figure 6.2: The rule of thumb refinement, halving the innermost elements at each refining step

Finally we did the T-spline refinement scheme. This was in many ways identical to the adaptive NURBS scheme. We solved the system of equations to get an approximate solution which we then estimated the error on. The only thing which was conceptually done differently was the refining itself. While there was some obviously different details which had to be taken care of with respect to the assembly of $\boldsymbol{A}$ and evaluation of the basis functions, it was in essence carried out by the exact same structure. The refinement however utilized a T-spline refinement scheme. This detected the elements in which the largest error was present followed by a refinement call on this element. Since it was possible that the element was in the middle of a T-mesh face, we had to search for the nearest T-mesh edge which we could then split in two by the refinement algorithm described in appendix B. This would effectively create more T-spline functions which could be used to represent a more locally accurate solution, and also split the element in two. The results of continued refinement is displayed in figure 6.1-6.1. The plot is here showing both the T-mesh mapped to the physical domain and the elements over which numerical integration is performed. As we can see there are more elements than there are T-mesh faces. This is however to be expected, and we also see that the T-spline elements are being spanned from a limited range from the finest T-mesh. This allows for the number of elements to scale as the number of T-mesh faces. We also note that there exist L-joints in figure 6.6a, but the corresponding element mesh in figure 6.6b is only

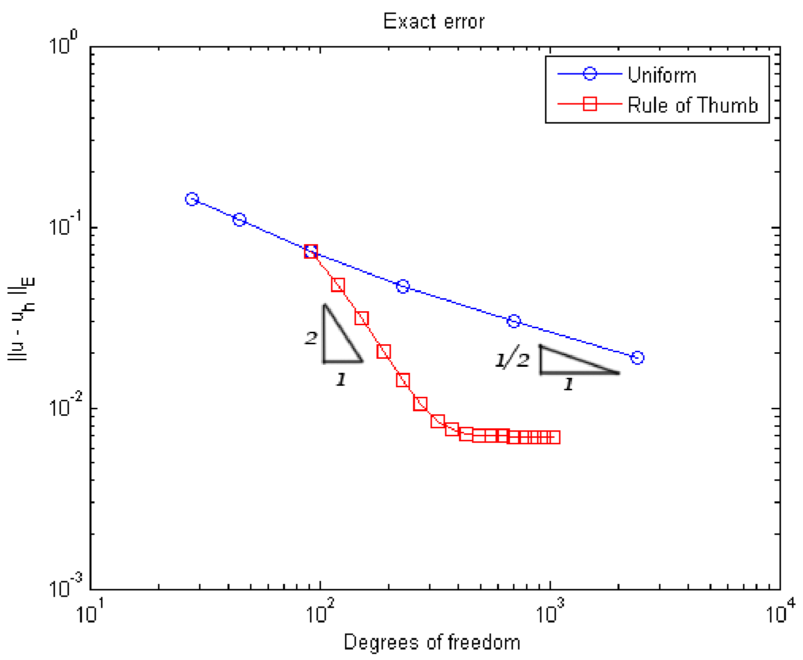


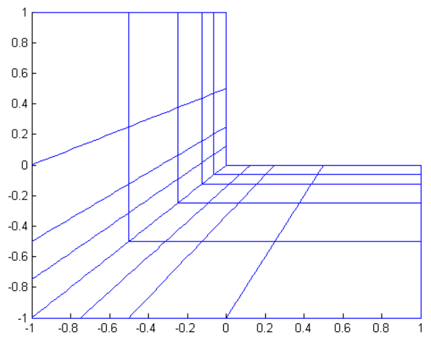
Figure 6.3: The convergence rates as a function of the degrees of freedom

consisting of rectangles.

For comparison we have included the convergence rates of all four refinement schemes in figure 6.7. Here we clearly see that the T-splines perform superior compared to the other methods.

6.2 Linear elasticity

We also did the linear elasticity problem with T-splines. With this case we didn't have any singularity as we did with the L-shaped problem. This resulted in a more evenly spread refinement with a small biasing towards the center of the hole. This is due to the fact that the exact solution has the most changes around this point. We did two refinement schemes here, uniform and T-spline. What was interesting in the uniform refinement was that the refinements seemed to be less than uniform in the physical domain. This was due to the parametrization as already discussed in the previous section. This is pictured in figure 6.2, where we have drawn the parametric and physical space of the domain, along with the control mesh. As we can clearly see, the refinement which is uniform in the parametric space is biasing towards to upper right corner in the physical space. This phenomena is even more apparent in figure 6.9 where we have more refinements. While it is possible to override this by manually choosing a different refinement scheme in the parametric domain (which would cause uniform refinement at the north and east


(a) Adaptive refinement scheme at an intermediate step

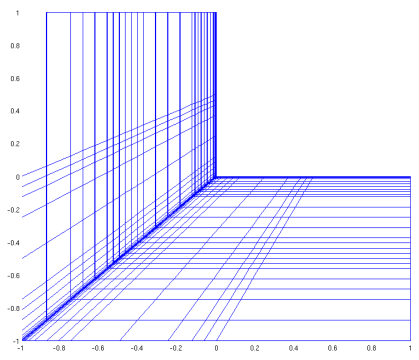
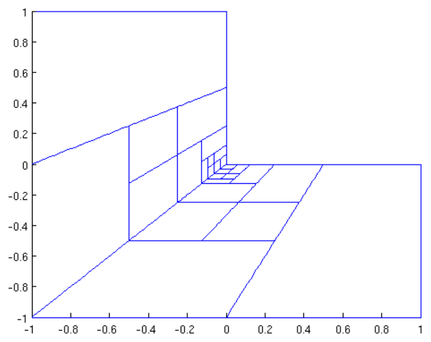

(b) Adaptive refinement at a late stage in the solution process

Figure 6.4: Adaptive refinements


(a) The T-spline mesh

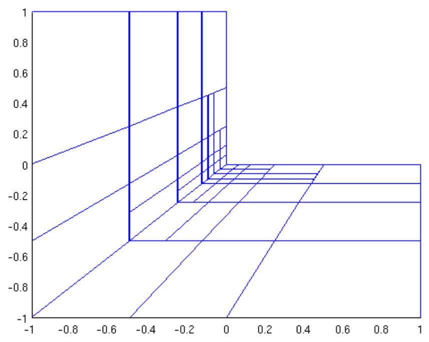
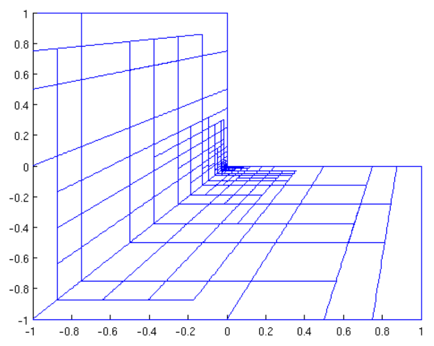

(b) The T-spline elements

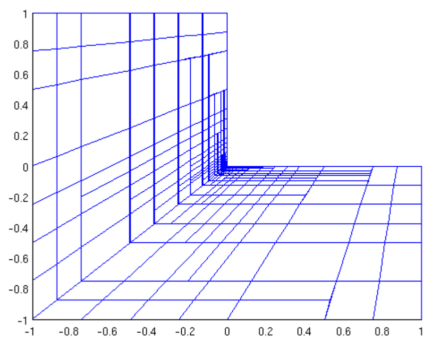
Figure 6.5: T-spline refinements at an intermediate step

edge for the cost of non-uniform refinements at the inner circle), we will not choose this approach. Instead we will let this refinement scheme stand, and note that this might not be the best parametrization which we could have chosen.

When choosing to refine by T-splines something interesting happened. While it did what it was supposed to do with cluttering enough blending functions around the inner circle it proved to have some other properties as well. Due to the large areas which was created at the top left and lower right or the domain (see figure 6.8c), these areas had large error contributions. The algorithm did then detect this and refine them. What was not refined as much was the upper right corner since that area received so many contributions from other places. This meant that the T-splines actually *negated* the effect of the bad parametrization. There was an on average uniform refinement taking place outside a small border from the inner circle. We see from figure 6.2 that the T-spline is favoring a uniform refinement. There are three major clutterings of points in



(a) The T-spline mesh



(b) The T-spline elements

Figure 6.6: T-spline refinements at a late stage

the parametric space. The first is the bottom border $\eta = 0$ which corresponds to the inner circular arch. The second and third is the upper right and upper left corner. The reason for such fine refinement here is this area which is mapping to the lower right and upper left of the physical space. And from the previous discussion we saw that it was exactly this area which was subject to the bad parametrization problems.

Lastly we take a look at the convergence rates of the uniform refinement strategy vs the T-spline refinements in figure 6.11. T-splines does yet again out-perform the uniform refinement, though not as definite as in the case of the L-shaped problem. This is because a true uniform refinement is not far from optimal in the case of the hole problem. The major sources of difference are a slight preference of refining around the inner circle, and the bad parametrization canceled by the T-splines.

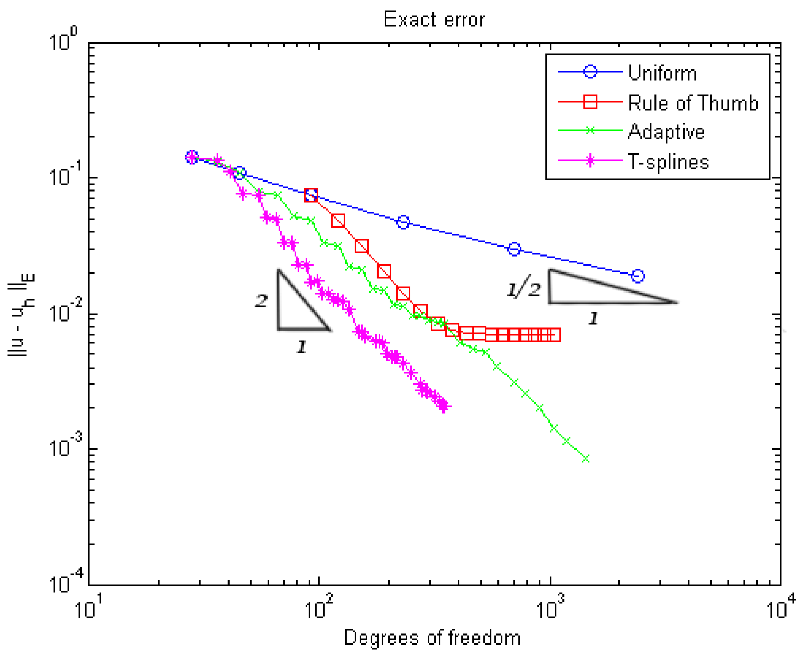
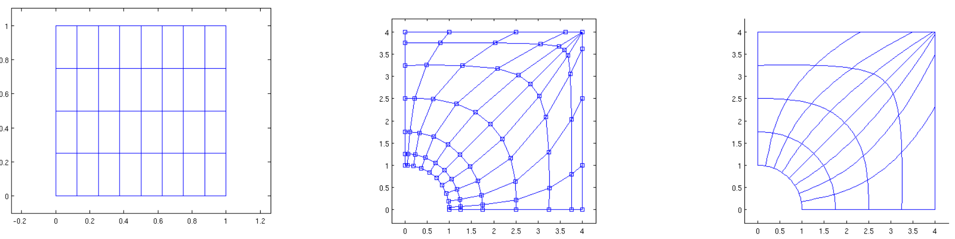


Figure 6.7: The convergence rates of all four refinement schemes



(a) Uniform refinement in the parametric space

(b) The control mesh used to generate the mapping

(c) Domain in the physical space. Refinement is not so uniform anymore

Figure 6.8: Uniform refinement problems

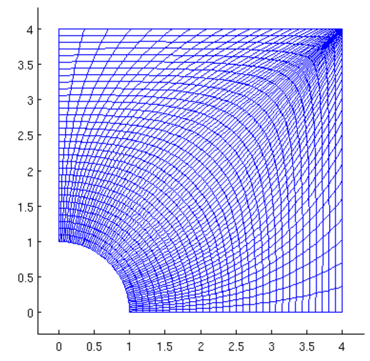
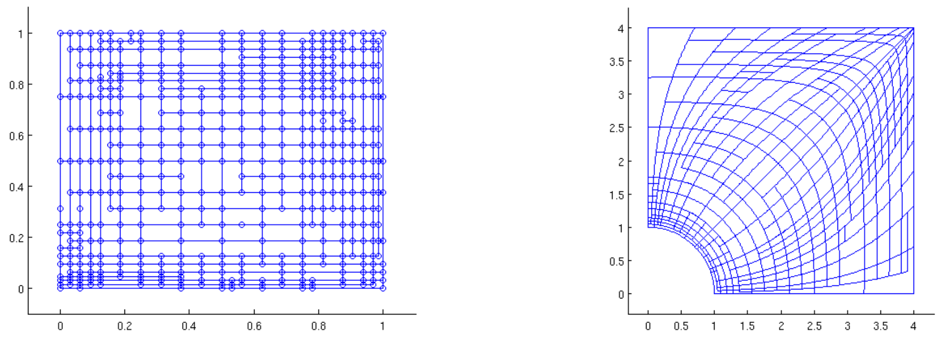


Figure 6.9: Uniform refinements at a late stage in the solution process



(a) Parameter space

(b) Physical space

Figure 6.10: T-spline mesh after a series of refinements

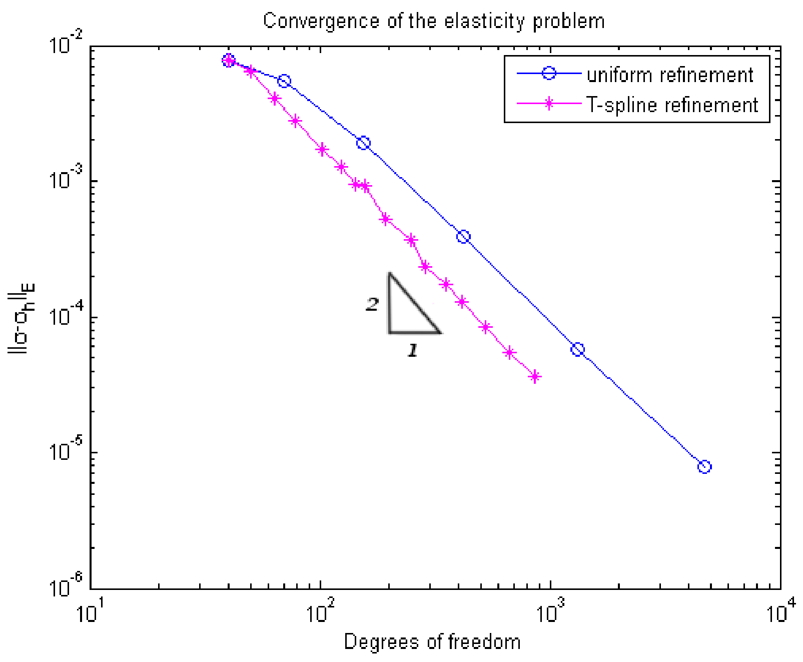


Figure 6.11: Convergence rates of the uniform vs T-spline refinement on the hole problem

Chapter 7

Discussion

7.1 Conclusions

We have shown that T-splines are realistic choices of basis functions in an isogeometric environment. Their performance is superior in cases where local refinement is needed, and they are the natural expansion of B-splines and NURBS. Not only do they allow for true local refinement as opposed to the tensor product nature of the NURBS, but they possess other remarkable properties as well. While refining a badly parameterized model, T-splines combined with a posteriori error estimation detected this and responded by clustering up the refinements in the parametric space where needed. This is a huge advantage as it enables the designer to focus on the task at hand, which is creating the model in the first place. Whether or not it is optimized for numerical analysis should be of less concern as this should be handled automatically by the adaptive algorithm. Remembering from the introduction that the creation of a model in which numerical analysis is *possible* takes up to 20 percent of the total analysis time. Making a model which is *good* for analysis takes up to 60 percent of total analysis time. While the isogeometric paradigm itself makes the first step superfluous, T-splines, along with adaptive mesh refinement schemes is what is making it possible to automate this last step.

T-splines are seeing their way into the industry, with several leading CAD design tools such as Rhino$^{\text{TM}}$ and Maya$^{\text{TM}}$ implementing them through a plugin. Being a superset of NURBS they are quite possible to coexist side by side with existing NURBS methods and programs. Both of these properties are crucial for industrial success. It also allows for complete backward compatibility in both an analysis and design setting.

7.2 Future work

We present several algorithms in this thesis that still leave many details to the imagination of the implementer. These include the T-mesh local knot vector extraction and T-spline refinement schemes. At several points searches for the closest point or intersecting lines are performed. While it is perfectly *possible* to brute force these things, which is to perform an all-to-all search, this will not be sufficient in the long run. Algorithms that scale well are needed. Effective search algorithms or indexing will need to be established along with corresponding datastructures. The problem of efficient T-spline implementation has so far been vacant in the literature.

T-splines is a relatively young research topic and there are several unsolved problems related to this. It is still not known if the T-spline blending functions are linearly independent. In fact, there is still some gray areas in the T-spline community on exactly what kind of T-spline topologies are to be allowed. The question of linear independence might be dependent on the existence of L-joints, I-joints or similar exotic T-mesh configurations. Currently, all indications lead us to believe that the T-mesh is indeed linearly independent, but a formal proof has yet to be published. This question would have a huge impact on the continued use of T-splines in finite element methods.

Another unsolved problem in the T-spline technology is the conversion from surfaces to solids. CAD-systems are exclusively representing object *surfaces*, while any analysis framework will usually require object *solids*. This will open the problem of how to automate the process of creating a solid object representation from its outer surface representation.

The error estimator used in this thesis is not among the most sophisticated available. While perfectly suited for its use in this thesis, it has several flaws. The field of a posteriori error estimation has become a mature field, with very rich literature. Among these are not only the residual-based estimator as used here, but also recovery-based and extraction-based estimators [17]. It might prove interesting to see how these different estimators perform with spline basis functions, especially with the observation that you can very easily create a continuous flux solution which will remove any error contributions from the C^0 element edges. It would also allow for approximation of the actual numerical value of the error itself, and not just the relative difference of the error in between the elements.

Appendix A

Evaluation of the B-Spline basis function

The B-spline basis functions are not only appearing in B-splines, but also NURBS, T-splines and PB-splines to name a few. Being the basis function it is obvious that they will be evaluated quite a few times during any FEM solution algorithm. It is therefore essential that it is possible to evaluate them in an efficient way. As it turns out, there exists an efficient algorithm for this problem, and I will give an overview of it here. The basis functions are defined as,

$$N_{i,p}(\xi) = \frac{\xi - \xi_i}{\xi_{i+p} - \xi_i} N_{i,p-1}(\xi) + \frac{\xi_{i+p+1} - \xi}{\xi_{i+p+1} - \xi_{i+1}} N_{i+1,p-1}(\xi), \tag{A.1}$$

and,

$$N_{i,0}(\xi) = \begin{cases} 1 & \text{if} \quad \xi \in [\xi_i, \xi_{i+1}) \\ 0 & \text{else} \end{cases} \tag{A.2}$$

with the exception of zero denominator where the entire fraction is evaluated to be zero. This special case is assumed to be handled by the algorithm at the appropriate locations and to increase readability I will not specify this in the algorithm description.

Now it is quite possible to evaluate (A.1) as it stands, that is by creating an recursive algorithm which takes the parameters i, p, ξ and Ξ and returns the value simply by calling itself recursively. This is described in detail in procedure 3. However, this will not be an efficient implementation. To see this I have drawn the function calls for a particular evaluation $N_{3,3}(\xi)$ in figure A.1 where I have abbreviated `N(i,p)` for the function signature since both the ξ- and Ξ parameter will be constant throughout the sequence. As is seen, `N(3,3)` will call `N` with parameters `N(3,2)` and `N(4,2)`. `N(3,2)` will then be recursively calling the function with respectively (3,1) and (4,1) as the parameters, etc. until the calling sequence is interrupted when it reaches $p = 0$ in which it is then evaluated using (A.2).As is illustrated in figure A.1 there is redundant work since the function will be called multiple times using the exact same argument. The yellow nodes all mark places where the function `N` is evaluated using the arguments (4,0)

Procedure 3 BSpline_Recursive(i, p, ξ, Ξ)

```
if  $p = 0$  then
    if  $\xi_i < \xi < \xi_{i+1}$  then
        return 1
    else
        return 0
    end if
else
    $N \leftarrow \frac{\xi-\xi_i}{\xi_{i+p}-\xi_i}\cdot$ BSpline_Recursive($i, p-1, \xi, \Xi$)
    $N \leftarrow N + \frac{\xi_{i+p+1}-\xi}{\xi_{i+p+1}-\xi_{i+1}}\cdot$ BSpline_Recursive($i+1, p-1, \xi, \Xi$)
    return $N$
end if
```

which obviously is a waste of computational power. This is a classical flaw of recursive algorithms, but one that also has a solution.

The running-time of the recursive algorithm was $\mathcal{O}(2^p)$ which is exponential in problem size. We will exchange this to $\mathcal{O}(p^2)$, which is only polynomial, by using dynamic programming. A full introduction of dynamic programming is beyond the scope of this work, but details can be found in any elementary book on algorithm construction, i.e. Cormen [7]. The basic idea is to solve the system "bottom-up" instead of "top-to-bottom". While the recursive definition starts by asking the question of what is the solution to `N(3,3)` the dynamic programming approach starts by asking what is the answer to `N(0,0)` and builds it solution space up larger and larger until the answer to the original problem has been found.

The first observation we do on the way to the final solution algorithm is the already hinted at in figure A.1 which is the fact that the solution for `N(i,p)` is only dependent on the solution of itself for a lower p. The knot vector Ξ and the evaluation point ξ will need to be passed into the algorithm, and are used to evaluate the function for specific i and p but are not affected by the recursion and dependencies which makes the dynamic programming work.

Since it is trivial to solve the problem for any $p = 0$, this is our starting point. In figure A.2 this has been marked as the first row. We will solve the problem for all i and $p = 0$ using (A.2). We will then extend this solution by calculating the solution for all i, and $p = 1$, using the fact that we already know the solution to the $p = 0$-problem. This will then spawn the solution of the $p = 2$-problem etc. By creating a $p \times n$-matrix we can gradually fill this up with the solutions. For any arbitrary point in the solution matrix, the evaluation of `N(k,q)` is only dependent on the two already known solutions `N(k,q-1)` and `N(k+1,q-1)`.

While this is fine by itself, it has only dropped the solution time of the algorithm to $\mathcal{O}(np)$ where n is the number of basis functions. Given that we know $p \ll n$ it is a significant improvement to realize that our solution-matrix will be sparse. Since we can already tell from the dependencies, we will not need to calculate all values, but rather

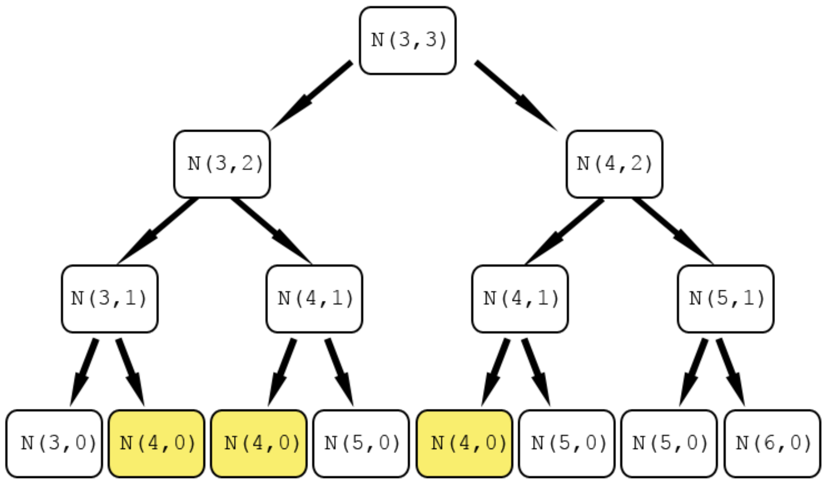


Figure A.1: Recursive function-calls

only those which are of any relevance to our desired value. As shown in figure A.3 this will only be a sub matrix of size $p \times p$.

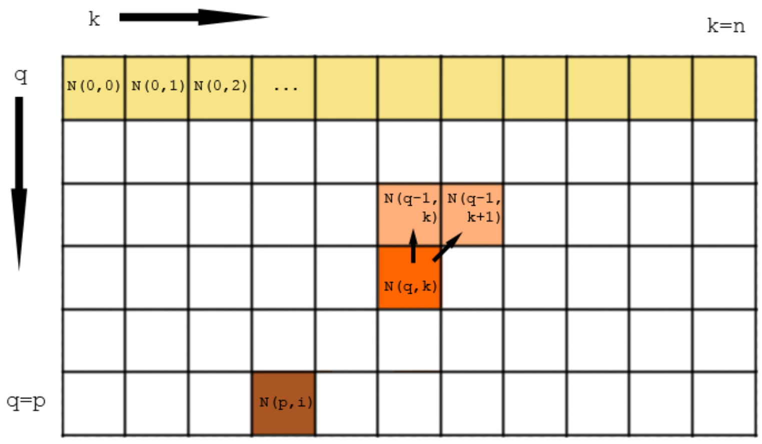


Figure A.2: Dynamic programming approach

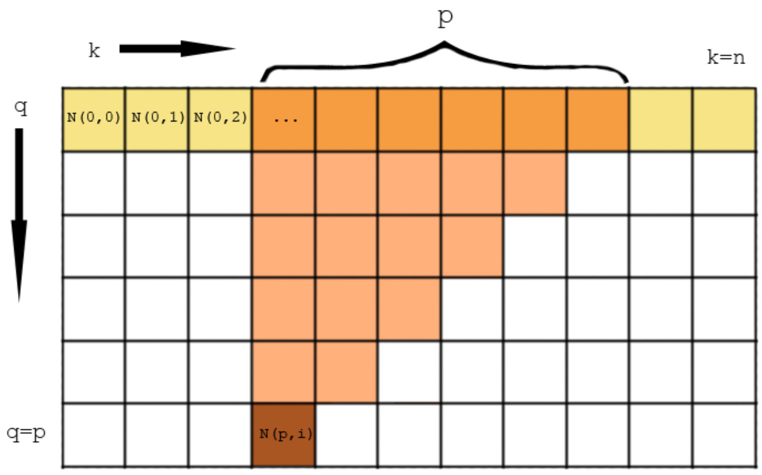


Figure A.3: The dependencies for one particular B-spline function

Appendix B

T-spline refinement

T-spline refinement is the act of increasing the number of blending functions, while still preserving the mapping of the surface unchanged in any way. To keep the refinement local this is done by a particular algorithm which we will attempt to describe here. The algorithm has some drawbacks in that it cannot insert any arbitrary T-mesh point without meeting some requirements. Usually these are the existence of nearby knot points. If these are not present, then they must first be created before the new function can be poperly inserted. So from an outside perspective you would usually request to have one or two new index knot points inserted, but you will receive a higher number, say five or six. The precise number of new points you will get is highly dependent on the T-mesh in question, but it is "usually" a quite small number. With the improved algorithm by Sederberg et al. (2004) [19] the number of additional index knot points which was inserted was reduced significantly.

B.1 Local splitting

T-spline refinement relies on the fact that B-splines can exactly be split into multiple functions. For the sake of simplicity, let us restrict ourselves to cubic splines. With $p = 3$, the local knot vectors will consist of five elements. Inserting a new knot ξ^* into the local knot vector $\Xi_k = [\xi_1, \xi_2, \xi_3, \xi_4, \xi_5]$ will split this into two blending functions. Depending on where the new point will need to be inserted these functions be one of the following

$$\begin{aligned}
N_k([\xi_1,\xi_2,\xi_3,\xi_4,\xi_5]) &= cN_k^1([\xi_1,\xi^*,\xi_2,\xi_3,\xi_4]) + dN_k^2([\xi^*,\xi_2,\xi_3,\xi_4,\xi_5]) \\
N_k([\xi_1,\xi_2,\xi_3,\xi_4,\xi_5]) &= cN_k^1([\xi_1,\xi_2,\xi^*,\xi_3,\xi_4]) + dN_k^2([\xi_2,\xi^*,\xi_3,\xi_4,\xi_5]) \\
N_k([\xi_1,\xi_2,\xi_3,\xi_4,\xi_5]) &= cN_k^1([\xi_1,\xi_2,\xi_3,\xi^*,\xi_4]) + dN_k^2([\xi_2,\xi_3,\xi^*,\xi_4,\xi_5]) \\
N_k([\xi_1,\xi_2,\xi_3,\xi_4,\xi_5]) &= cN_k^1([\xi_1,\xi_2,\xi_3,\xi_4,\xi^*]) + dN_k^2([\xi_2,\xi_3,\xi_4,\xi^*,\xi_5])
\end{aligned} \tag{B.1}$$

with

$$c = \begin{cases} 1 & \text{if } \xi^* < \xi_2 \\ \frac{\xi_5-\xi^*}{\xi_5-\xi_2} & \text{otherwise} \end{cases}$$

$$d = \begin{cases} 1 & \text{if } \xi^* > \xi_4 \\ \frac{\xi^*-\xi_1}{\xi_4-\xi_1} & \text{otherwise} \end{cases}$$

For notational ease, we shall define the parent knot vector to be the vector composed of the nondecreasing set (of size six) $\tilde{\Xi_k} = \Xi_k \cup \xi^*$. The blending function Ξ_k will then be split into two functions, one which is composed of the first five elements $\tilde{\Xi_k^1}$, and one of the five latter $\tilde{\Xi_k^2}$.

This is a known equality, and is also the basics for regular B-spline refinement. It is the only mathematical identity required to perform the following refinements. By continued use of this at chosen points, we are going to do the entire refining process. When it comes to blending functions defined over two variables (ξ, η), the function splitting will split one of the parametric functions, and keep the other unchanged. If we were to insert a knot ξ^* into the function $R(\Xi_k, \mathcal{H}_k)$ then this would be

$$\begin{aligned} R(\Xi_k, \mathcal{H}_k) &= N(\Xi_k)N(\mathcal{H}_k) \\ &= \left(cN_k^1(\tilde{\Xi_k^1}) + dN_k^2(\tilde{\Xi_k^2})\right) N(\mathcal{H}_k) \\ &= cR_k^1(\tilde{\Xi_k^1}, \mathcal{H}_k) + dR_k^2(\tilde{\Xi_k^2}, \mathcal{H}_k). \end{aligned}$$

Inserting one control point in a two-dimensional function $R(\xi, \eta)$ can thus be done by splitting the one-dimensional problem and keeping the other parametric knot vector unchanged. We could then proceed by splitting the two new functions R_k^1 and R_k^2 by inserting more knots in either the ξ- or η direction.

As an example of how the splitting equations can be used, we will take a look at figure B.1 where we will split the blending function located at P_1 into four blending functions centered at P_2, P_3 and P_4. This is done by continued use of the splitting functions (B.1).
When we say "split" the functions, it is to be understood that the coefficients c and d in (B.1) are to be multiplied with the corresponding control points to each blending function. After all, it is the physical surface $\boldsymbol{C}$, given by the blending functions multiplied by the control points which we will wish to preserve intact. So to get back to the example at hand, we will have 4 control points $\boldsymbol{B}_i$ corresponding to the four blending functions. Since P_2, P_3 and P_4 does not exist prior to the splitting, we will have that the local knot vector to R_1 is $\Xi_1 = [\xi_1, \xi_2, \xi_3, \xi_5, \xi_6]$ and $\mathcal{H} = [\eta_1, \eta_2, \eta_4, \eta_5, \eta_6]$. This can then be split into R_1 and R_2 by

$$\begin{aligned} R_1[\xi_1, \xi_2, \xi_3, \xi_5, \xi_6][\eta_1, \eta_2, \eta_3, \eta_5, \eta_6] &= cR_1[\xi_1, \xi_2, \xi_3, \xi_4, \xi_5][\eta_1, \eta_2, \eta_3, \eta_5, \eta_6] \\ &\quad +dR_2[\xi_2, \xi_3, \xi_4, \xi_5, \xi_6][\eta_1, \eta_2, \eta_3, \eta_5, \eta_6] \quad \text{(B.2)} \end{aligned}$$

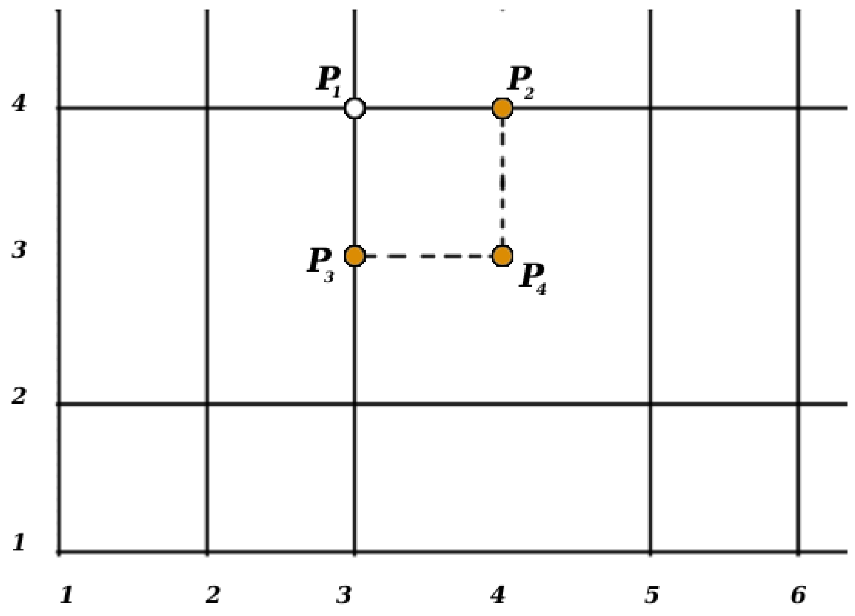


Figure B.1: An example splitting of P_1 into P_2, P_3 and P_4

We now see that we have another local knot vector Ξ_1 for the blending function centered at R_1. We will split this new function by

$$\begin{aligned} R_1[\xi_1,\xi_2,\xi_3,\xi_4,\xi_5][\eta_1,\eta_2,\eta_3,\eta_5,\eta_6] &= cR_3[\xi_1,\xi_2,\xi_3,\xi_4,\xi_5][\eta_1,\eta_2,\eta_3,\eta_4,\eta_5] \\ &\quad + dR_1[\xi_1,\xi_2,\xi_3,\xi_4,\xi_5][\eta_2,\eta_3,\eta_4,\eta_5,\eta_6] \quad \text{(B.3)} \end{aligned}$$

Lastly we will split the function R_2 into the two functions centered at R_2 and R_4 by

$$\begin{aligned} R_2[\xi_2,\xi_3,\xi_4,\xi_5,\xi_6][\eta_1,\eta_2,\eta_3,\eta_5,\eta_6] &= cR_4[\xi_2,\xi_3,\xi_4,\xi_5,\xi_6][\eta_1,\eta_2,\eta_3,\eta_4,\eta_5] \\ &\quad + dR_2[\xi_2,\xi_3,\xi_4,\xi_5,\xi_6][\eta_2,\eta_3,\eta_4,\eta_5,\eta_6] \quad \text{(B.4)} \end{aligned}$$

If we now combine (B.2)-(B.4) we can write this as

$$R_1^*[\xi_1,\xi_2,\xi_3,\xi_5,\xi_6][\eta_1,\eta_2,\eta_3,\eta_5,\eta_6] = c_1P_1 + c_2P_2 + c_3P_3 + c_4P_4 \quad \text{(B.5)}$$

where R_1^* is the original blending function, c_i can be given explicitly and the blending functions R_i on the right hand side are all dependent on the correct local knot vectors at the end of the splitting. If we were to implement this splitting, we would only need to update the control points associated with each blending function. That is

$$\begin{aligned} \boldsymbol{B}_1 &= c_1\boldsymbol{B}_1^* \\ \boldsymbol{B}_2 &= c_2\boldsymbol{B}_1^* \\ \boldsymbol{B}_3 &= c_3\boldsymbol{B}_1^* \\ \boldsymbol{B}_4 &= c_4\boldsymbol{B}_1^*. \end{aligned}$$

B.2 The violation

One of the most crucial things to understand about the T-spline refining algorithm is that it can violate many rules set by T-splines during the refining process. Through out the algorithm execution the T-spline may violate several things and take on strange forms. At the entry of the algorithm it should be a valid T-spline, and the goal of the algorithm is to make it a valid T-spline at the end of it as well. Before any insertions of any knot index points, we store every local knot vector corresponding to each point. These local knot vectors will be manipulated throughout the algorithm. The violations which may take place are the following

Violation 1 A blending function is dependent on a nonexistent knot.

Violation 2 A blending function has dependencies which stretch farther than dictated by algorithm 1.

Violation 3 There exist multiple knot index points at the same location

As already referred to, the knot vector extraction given in algorithm 1 plays a central part in the refining algorithm. The algorithm will start by inserting the requested points. These will interfere with the previous results for the local knot vector extraction, and several violations will have been introduced. We will resolve these violations, one at a time until every local knot vector is corresponding to the local knot vector as dictated by the local knot vector extraction algorithm.

What we will do is classify each violation, and then resolve it by the appropriate steps. Firstly we will extract the local knot vectors as dictated by the rules of the T-mesh, i.e. by algorithm 1. These are dubbed $\hat{\Xi}$

Violation 2 happens when we are inserting a point or a line which is passing through the middle of another blending functions support. This will have the old index knot point detecting a local knot vector which is dependent on points closer to the point itself. The violation is resolved by splitting the violating blending function into two functions with different support by (B.1).

In the case of violation 1 this cannot be resolved by any splitting of existing functions. It is typically happening when a function is dragging along its second nonsplitted local knot vector to the new point. In this case we have no option other than to insert a brand new index knot point at the appropriate location and create this with dummy values $(0, 0, 0)$.

The solutions to both of the above violations will create new blending functions. Violation 1 is creating a zero-initialized function, and violation 2 is creating one new blending function and altering the old one. In both cases there are the creation of one new function. This function can be placed anywhere, including on top of existing functions. Since the local knot vectors can deviate from those enforced by the T-mesh rules during the refining algorithm we have no guarantee that the blending functions from points lying on top of each other are the same. At the point where we have eliminated all of violation 1 and 2 however we know this. At that point we are resolving violation 3 by merging all points which lie on top of each other into one single blending function.

B.3 Illustrated example

To illustrate some of the violations along with their solution, we will give an illustrated example of one knot insertion. We will start with a valid T-spline and request the addition of one more index knot point. As we will see, the algorithm will require that we actually insert two. It is shown in figure B.3.

In (a) we are requesting the insertion of the index knot point (4,4). This is then added with a zero control-point initializer.

In (b) we see that there are four control points which are causing violation 2. The local knot vector of the point at (4,5) is drawn, it is $\Xi_{(4,5)} = [\xi_1, \xi_2, \xi_4, \xi_5, \xi_6]$ and $\mathcal{H}_{(4,5)} = [\eta_2, \eta_3, \eta_5, \eta_6, \eta_7]$ which was correct *before* the new knot was inserted. However now it's stored knot-span is too large and we will need to split it's $\mathcal{H}$-vector into two with smaller support.

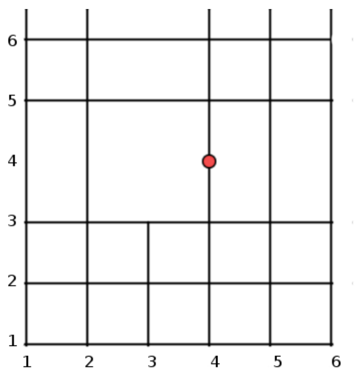
In (c) the point at (4,5) is successfully split by (B.1). This is causing the creation of to functions $\mathcal{H}_{(4,5)} = [\eta_3, \eta_4, \eta_5, \eta_6, \eta_7]$ and $\mathcal{H}_{(4,4)} = [\eta_2, \eta_3, \eta_4, \eta_5, \eta_6]$. They will share the same Ξ vector. After this the exact same thing is done by splitting the point at (4,6) into two points.

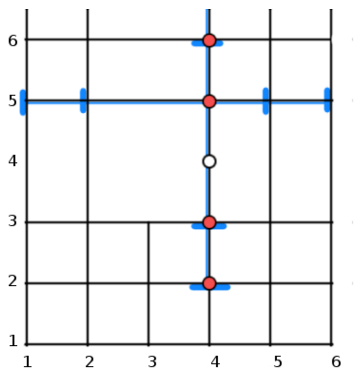
In (d) we see that we have gotten two points at (4,4) already. They are from the two previous splittings. We now proceed to split the one centered at (4,3). What is interesting to note is that this splitting will bring along another knot vector Ξ than the previous too splittings did (see (c)). The point at (4,2) is also split, totaling four new points at (4,4).

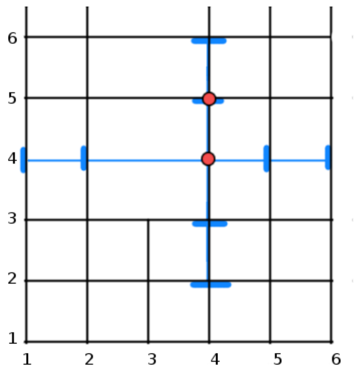
In (e) we see the first entry of violation 1. Here the points which was split in (d) had a Ξ-vector which dictated an entry at ξ_3. However there was no points or lines to suggest this in the mesh. It is therefore needed to insert one. We are inserting a new index knot point at (3,4) and adding a zero control-point initializer. Following this we see that the two points created at (c) are in violation 2. They are both split resulting in two new points at (3,4).

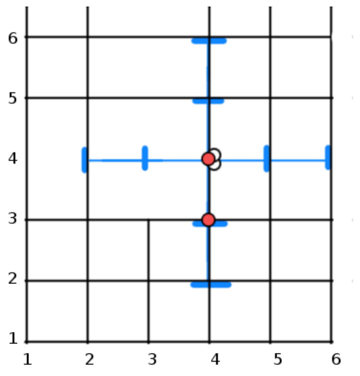
In (f) there is two more cases of violation 2. Both the point at (3,2) and (3,3) will need to be split into points centered at (3,4), totaling four new points at that location. After this, there are no more violation 1 or 2. However there are violation 3 as there are multiple points at both (3,4) and (4,4).

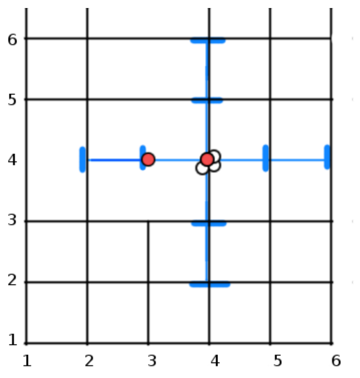
In (g) the multiple points are merged, simply by adding their control points together. After this we are creating all the new horizontal or vertical lines which may be added. The insertion is complete and we are left with a valid T-spline which has not altered it's shape in any way. We have gotten two more points and two more lines, even if we only requested one point.

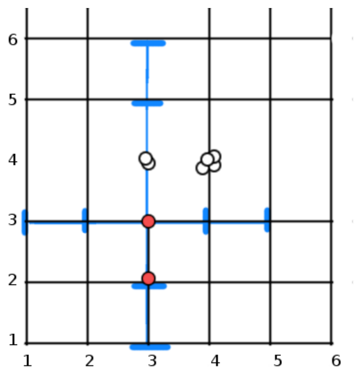

(a) New red point requested


(b) 4 points causing violation 2


(c) The two upper points are split keeping their Ξ-vector


(d) Bottom points are split keeping *their* Ξ vector


(e) The new point from is causing violation 1


(f) More violations 2

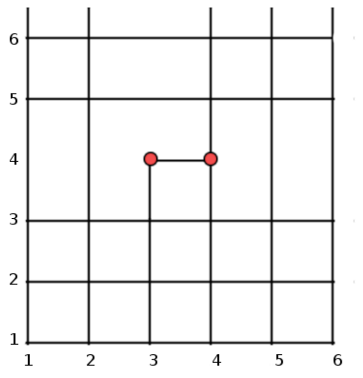

(g) Merging points and drawing lines

Figure B.2: A visual T-spline insertion example

B.4 Numerical example

Fortunately when implementing this, there is a very simple test to see if a particular blending function is violating violation 1 or 2. Typically you would work with two local knot vectors. You would store the local knot vector Ξ corresponding to each index knot point, and whenever we suspect that there might be a violation, we extract the local knot vector $\hat{\Xi}$ as dictated by the T-mesh rules. If they are equal, then we have no violation. If they are not equal, then we proceed with finding which knot is responsible for this. If the detected knot $\hat{\xi_i}$ is further away from the index knot point that what the stored knot ξ_i is, then this corresponds to violation 1 and we will have to insert an additional index knot point at the needed point ξ_i. This test can be implemented as $|\hat{\xi_i} - \xi_3| > |\xi_i - \xi_3|$ assuming that we are dealing with cubic splines and the knot vector size is 5 such that the middle point is ξ_3. Remember that every local knot vector has stored its center of origin at the middle of its knot vectors. The other case is of course violation 2, which is that the stored knot vector is too large, and we will need to split it into smaller ones. This happens if the detected knot vector $\hat{\Xi}$ contains knots which are closer to the index knot point than what is stored, i.e. $|\hat{\xi_i} - \xi_3| < |\xi_i - \xi_3|$. The appropriate response to this is splitting around the newly detected $\hat{\xi_i}$. This means inserting $\hat{\xi_i} = \xi^*$ in (B.1). The argument follows exactly for the $\mathcal{H}$ knot vector, and they can be treated completely decoupled and in any order we wish.

To give an numerical example on how we may get the input consider the case that we had a knot vector Ξ and the T-mesh rules stated that the knot vector should be $\hat{\Xi}$ as given in (B.6).

$$\begin{aligned}\hat{\Xi} &= [1, 2, 5, 6, 7]\\ \Xi &= [3, 4, 5, 8, 9]\end{aligned} \tag{B.6}$$

Note that the middle point is equal as it will always be since it is the index knot point itself. Obviously $\hat{\Xi} \neq \Xi$ so violations exist. We will proceed with finding them, classifying them and resolving them. Starting from an arbitrary position, we note that $\hat{\xi_2} = 2$ is farther away from the point itself $\xi_3 = 5$ than it's stored equivalent $\xi_2 = 4$. This is thus violation 1, and we will need to insert a new point. The new point is the one detected which is 4. After inserting this, then we are obviously guaranteed to detect this in the next detection step. The new case will then be

$$\begin{aligned}\hat{\Xi} &= [2, 4, 5, 6, 7]\\ \Xi &= [3, 4, 5, 8, 9]\end{aligned}$$

This blending function is still in violation, and we are now resolving the case $\xi_5 \neq \hat{\xi_5}$. Since the detected 7 is closer to the point itself than the stored 9, we will need to split this function. We will split it around the detected 7. According to (B.1) with $\xi^* = 7$ this results in two blending functions: $\Xi_1 = [3, 4, 5, 7, 8]$ and $\Xi_2 = [4, 5, 7, 8, 9]$. Of these, only Ξ_1 is centered at the same point as we initially started, so this is the only one which we will be fixing. The other blending function is probably in violation as well, but will be handled at a later point in the algorithm. The new knot vectors are

$$\hat{\Xi} = [2, 4, 5, 6, 7]$$
$$\Xi = [3, 4, 5, 7, 8]$$

We will now proceed with fixing the first knot. 2 is further from the center than 3, so this is violation 1, which is handled by inserting the point 3. This will then be subsequently be detected in the next sweep.

$$\hat{\Xi} = [3, 4, 5, 6, 7]$$
$$\Xi = [3, 4, 5, 7, 8]$$

Where only one knot remains different, which is the knot $\hat{\xi}_4 = 6$. This is closer than 7, which is a violation 2 and we split the existing knot into two by inserting 6. This results once again in two knot vectors $\Xi_1 = [3, 4, 5, 6, 7]$ and $\Xi_2 = [4, 5, 6, 7, 8]$. The first will we proceed with, while the second will be handled at a later point. But now we are finished since the detected knot is equal to the stored knot. We may now proceed with either fixing the $\mathcal{H}$-vector or some of the new points which was created by splitting existing points.

B.5 The algorithm

The algorithm is quite simple once you get used to the splitting itself. It is given in pseudo-code in algorithm 4

Algorithm 4 T-spline knot insertion

```
1: insert all newly requested knot points
2: repeat
3:    while there exist $p$ such that $\Xi_p \neq \hat{\Xi}_p$ do
4:       if $|\hat{\xi}_i - \xi_3| > |\xi_i - \xi_3|$ then
5:          create new point at $\xi_i$ with $\boldsymbol{B} = [0, 0, 0]$ {Violation 1}
6:       else if $|\hat{\xi}_i - \xi_3| < |\xi_i - \xi_3|$ then
7:          Split the knot using (B.1) with $\xi^* = \hat{\xi}_i$ {Violation 2}
8:       end if
9:    end while
10:   merge every duplicate index knot point
11:   draw every possible horizontal or vertical nonintersecting lines
12: until no changes are made
```

There is just a few comments which should be said last. The lines seem to be added at a slightly unnatural point. They do have a function, but this function is primarily for the user to have a more intuitively understanding of the grid in question. This is of course *very* important in a design setting, but it would also be interesting to explore the possibilities of refining the T-mesh without this step. Of course they have the function that they are restricting the support of other blending functions, which is a desired property it may not be advantageous to skip them altogether. This is some of the reason for the outer repeat-until loop. The addition of the lines may interfere with some of the existing blending functions, causing them in a state of violation 2. It is therefore necessary to repeat this process until we are sure that all points have knot vectors dictated by the T-mesh rules as well no more lines have been inserted.

Bibliography

[1] Mark Ainsworth and J. Tinsley Oden. A posteriori error estimation in finite element analysis. *Computer Methods in Applied Mechanics and Engineering*, 142(1-2):1 – 88, 1997.

[2] Mark Ainsworth and Tinsley J. Oden. *A Posterori Error Estimation in Finite Element Analysis*. Wiley-Interscience, 1st edition, January 2000.

[3] Y. Bazilevs, L. Beirao de Veiga, J. A. Cottrell, T. J. R. Hughes, and G. Sangalli. Isogeometric analysis: approzimation, stability and error estimates for h-refined meshes. *Mathematical Models and Methods in Applied Sciences*, 16:1031–1090, 2006a.

[4] Y. Bazilevs, V.M Calu, J.A. Cottrell, J.A. Evans, T.J.R. Hughes, S. Lipton, M.A. Scott, and T.W. Sederberg. Isogeometric analysis using t-splines. *Preprint submitted to Elsevier Science*, August 2008.

[5] Dietrich Braess. *Finite elements - theory, fast solvers, and applications in solid mechanics*. Cambridge University Press, third edition, 2007.

[6] R. L. Burden and J. D. Faires. *Numerical Analysis*. Thomson Brooks/Cole, eight edition, 2005.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

[8] J.A. Cottrell, T.J.R. Hughes, and Y. Bazilevs. *Isogeometric Analysis Toward Unification of CAD and FEA*. Institute for Computational Engineering and Sciences, The University of Texas at Austin, preprint edition, 2008.

[9] J.A. Cottrell, T.J.R. Hughes, and A. Reali. Studies of refinement and continuity in isogeometric structural analysis. *Computer Methods in Applied Mechanics and Engineering*, 196(41-44):4160 – 4183, 2007.

[10] M.R. Dörfel, Bert Jüttler, and Bernd Simeon. Adaptive isogeometric analysis by local h-refinement with t-splines. *Comput. Methods Appl. Mech. Engrg*, (2008).

[11] John B. Fraleigh. *A first course in Abstract Algebra*. Pearson Education, seventh edition, 2003.

[12] William J. Gordon and Charles A. Hall. Construction of curvilinear co-ordinate systems and applications to mesh generation. *International Journal for Numerical Methods in Engineering*, 7(4):461–477, 1973.

[13] Qi-Xing Huang, Shi-Min Hu, and Ralph R. Martin. Fast degree elevation and knot insertion for b-spline curves. *Comput. Aided Geom. Des.*, 22(2):183–197, 2005.

[14] T. J. R. Hughes, J. A. Cottrell, and Y. Bazilevs. Isogeometric analysis: Cad, finite elements, nurbs, exact geometry and mesh refinement. *Computer Methods in Applied Mechanics and Engineering*, 194(39-41):4135–4195, October 2005.

[15] T.J.R. Hughes, S. A. Reali, and G. Sangalli. Duality and unified analysis of discrete approximations in structural dynamics and wave propagation: Compariosn of p-method finite elements with k-method nurbs. *Preprint submitted to Elsevier Science*, October 2007.

[16] T.J.R. Hughes, S. A. Reali, and G. Sangalli. Efficient quadrature for nurbs-based isogeometric analysis. *Preprint submitted to Elsevier Science*, August 2008.

[17] Trond Kvamsdal and Knut Morten Okstad. Error estimation based on superconvergent patch recovery using statically admissable stress fields. *Int. J. Numer. Meth. Engng.*, 42:443–472, 1998.

[18] Richard Franklin Riesenfeld. *Applications of b-spline approximation to geometric problems of computer-aided design.* PhD thesis, Syracuse University, Syracuse, NY, USA, 1973.

[19] Thomas W. Sederberg, David L. Cardon, G. Thomas Finnigan, Nicholas S. North, Jianmin Zheng, and Tom Lyche. T-spline simplification and local refinement. *ACM Trans. Graph.*, 23(3):276–283, 2004.

[20] Thomas W. Sederberg, Jianmin Zheng, Almaz Bakenov, and Ahmad Nasri. T-splines and t-nurccs. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 477–484, New York, NY, USA, 2003. ACM.

[21] Kenneth James Versprille. *Computer-aided design applications of the rational b-spline approximation form.* PhD thesis, Syracuse University, Syracuse, NY, USA, 1975.

[22] J Wan, B Steele, SA Spicer, S Strohband, GR Feijóo, TJ Hughes, and CA Taylor. A one-dimensional finite element method for simulation-based medical planning for cardiovascular disease. *Comput Methods Biomech Biomed Engin*, 5(3):195–206, 2002.

[23] O. C. Zienkiewicz and J. Z. Zhu. The superconvergent patch recovery and a posteriori error estimates. part 1: The recovery technique. *International Journal for Numerical Methods in Engineering*, 33(7):1331–1364, 1992.