Lehrstuhl für Baustatik und Baudynamik

Univ.-Prof. Dr.-Ing. habil. S. Klinkel

Numerical Methods in Structural Mechanics and Dynamics

Quadtree Decomposition of Heterogeneous Materials in NURBS-boundary Representation

presented by:

Mario Aguilar Rueda

Matr.-Nr.: 383214

Benjamin Cerar

Matr.-Nr.: 399918

Hristo Kovachev

Matr.-Nr.: 307060

Supervisor:

Rainer Reichel, M.Sc. (LBB)

Aachen, March 31, 2019

Abstract

Composite materials have been for decades widely used in civil engineering and related fields. Some approaches for treating numerical models, which deal with different materials, involve the employment of uniform meshes. This might be expensive in terms of computational resources. High accuracy of the solution is required; therefore, a fine mesh is needed. At the same time, the number of elements produced by this kind of approaches must be reduced. This dilemma could be resolved by creating an automatic mesh that suits the geometry of the model, creating more elements only where it is needed.

A novel meshing technique for composite materials is presented. The description of interfaces is done with NURBS. This method makes use of the quadtree decomposition procedure, which generates non-uniform interface-conforming meshes. This allows us to create an automatic non-uniform mesh, based on the complexity of the geometry, and to maintain an exact representation of the boundary.

Contents

1 Introduction

Modern high-performance materials obtain their extraordinary properties from the interaction between their different components. The finer the distribution, the greater the importance of interfaces between the components. An example is fiber-reinforced concrete. The process of such development usually involves producing a computational model in order to simulate the heterogenous material behavior, and then verifying such models with experimental tests.

Developing a numerical model of a heterogeneous material presents a challenge due to the need to represent the boundary between different materials with a satisfactory degree of accuracy. In this project, the focus has been on developing a non-uniform mesh generator. NURBS are used for the mathematical description of the interfaces, and are taken as an input parameter, while quadtrees are implemented in order to create a non-uniform mesh.

The content herein is organized as follows. Firstly, the main theoretical concepts are introduced in Section 2. NURBS curves with some of their key properties are first defined. The quadtree procedure is discussed afterwards, where data tree structures and advantages of non-uniform meshes are also discussed. To conclude the theoretical part, a method to compute kernels of simple polygons is briefly introduced. Throughout the theoretical discussion, several algorithms to back up the concepts in question are also offered.

In Section 3, the implementation of the quadtree based mesh generator is presented. The explanations of the implementation are accompanied and exemplified with a representation of an interface between two different types of materials, being its shape designed for testing the different features of the code. First, a NURBS representation of this interface is displayed. Later on, the treatment of the NURBS during the decomposition is shown, while also an explanation for tree data structure organisation is given, and what information is stored during the quadtree decomposition. Two different kernel-finding algorithms follow. The procedure of quadtree balancing concludes this section.

In Section 4, a variety of NURBS-interface examples, where a quadtree decomposition of the domain is performed, is presented. It begins with some simple circumferential shapes, which are the most likely to be represented in composite materials. They could represent void or fiber sections. Afterwards, some flat-shaped geometries are studied, which could exemplify a fiber sheet. At the end, some more complex geometry is presented to test the functionality of the implementation.

2 Theoretical Background

2.1 NURBS Curves

The term NURBS stands for Non-Uniform Rational B-Splines. This is due to the NURBS being based on B-splines, therefore thesee are a logical starting point for their study [1]. They are commonly used in the modern CAD, CAE, and other computer design programs, because they can be implemented in a straight-forward manner, due to their mathematical definition. In the ensuing description of NURBS and their properties, we limit ourselves to one-dimensional curves, which are relevant for the problem at hand. The mathematical notation used in the definitions henceforth is taken from [2].

2.1.1 Definition of B-Splines

B-Splines are piece-wise polynomial functions, denoted as $\mathbf{C}(u)$, where u is a parameterical coordinate, based on which the curve is defined. A segment of the curve can be defined as $\mathbf{C}_i(u)$,$1 \leq i \leq m$, where n is the number of m-th degree polynomial segments.
Formally, a p-th degree B-Spline curve is defined as:

$$\mathbf{C}(u) = \sum_{i=0}^{n} N_{i,p}(u)\mathbf{P}_i \qquad a \leq u \leq b. \tag{2.1}$$

Above $\mathbf{P}i$ represents control points, which form a control polygon. $N_{i,p}(u)$ are the p-th grade B-Spline basis functions, defined at each breakpoint value u. a and b represent the limits of the parametrical domain, upon which the B-Spline basis functions, and consequently the curves, are defined. This space is defined by the knot vector U, and its elements are known as knots. The knot vector can be written in the following manner:

$$U = \{\underbrace{a, \ldots, a}_{p+1}, u_{p+1}, \ldots, u_{m-p-1}, \underbrace{b, \ldots, b}_{p+1}\}. \tag{2.2}$$

The basis function of degree *p* can then be defined as:

$$N_{i,p}(u) = \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u), \tag{2.3}$$

which is known as Cox–de Boor recursion formula. It can be seen from Equation (2.3), that the basis function is dependent upon lower-order basis functions. In fact, one can produce a triangular tree, consider, for example the basis function $N_{3,2}$. The lower-order basis functions, connected with it are:

$$
\begin{array}{ccc}
N_{3,0} & & \\
& N_{3,1} & \\
N_{4,0} & & N_{3,2} \\
& N_{4,1} & \\
N_{5,0} & &
\end{array}
$$

The zero-degree basis functions are discrete functions, and are defined as:

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

In Equation (2.2), it is implied that a knot value can occur several times, which influences several important properties of the resulting B-Spline. This is called knot multiplicity.
For the definition of B-Spline basis functions, it is also imperative that the order of the knot values is increasing. For instance, consider the knot vector $U = \{u_0, \ldots, u_m\}$. The knots must be ordered in a way so that $u_i < u_{i+1}$ for each $i = 0, \ldots, m$. An example of a knot vector, defined by twelve elements is:

$$\text{U} = \{0,0,0,0.25,0.25,0.5,0.5,0.75,0.75,1,1,1\}. \tag{2.4}$$

One can observe, that the first and last knots have a knot multiplicity of $p+1$, which means that the curve is "clamped" at those points. At the internal knots, we can see that that we have a C^0 continuity. That is due, as described, to having a knot multiplicity, which is equal to the degree of the curve. This can be observed as the discontinuity of B-Spline basis functions at internal knots, shown in Figure 2.1.

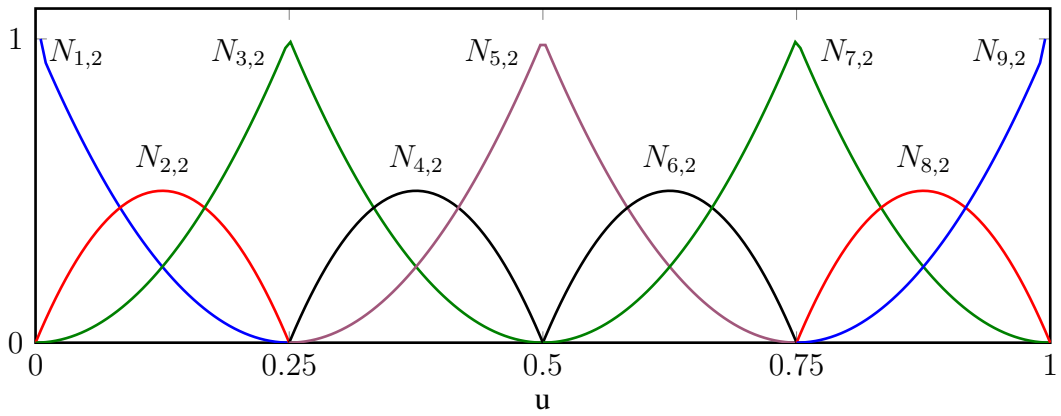


Figure 2.1: B-Spline basis function of the example curve

From Equation (2.3) it can be observed, that B-Spline basis functions are polynomials, but they are only defined on certain intervals of the knot vector, which we name knot spans. Due to this, B-Spline curves can be thought of as series of piece-wise polynomial functions.
In order to fully understand multiplicity of knots and its effects, derivatives of a B-Spline curve

must be introduced. The k-th derivative of the curve $\mathbf{C}(u)$ is defined as follows:

$$\mathbf{C}^{(k)}(u) = \sum_{i=0}^{n} N_{i,p}^{(k)}(u)\mathbf{P}_i \tag{2.5}$$

Based on this, the derivative of a B-Spline curve can be understood as the derivative of its corresponding basis functions. At a given knot u, a B-Spline basis function $N_{i,p}(u)$ is $p-k$ continuously differentiable, where k represent the multiplicity of the knot. Consequently, by increasing the multiplicity, the continuity of a curve can be decreased, which is an important concept in the principle of curve splitting, which is described later.

2.1.2 Properties of B-Splines

Based on the mathematical definition of B-Spline curves, shown in Equation (2.1) in Section 2.1.1, several properties that are relevant should be described.
As described, basis functions are only defined on certain segments of a knot vector. Formally, $N_{i,p} = 0$ when $u \notin [u_i,u_{i+p+1})$. From this it follows, that manipulating control points only affects a certain part of the B-Spline curve. Paritition of unity is a property, which can be derived from the definition of the basis functions: at any given knot span $[u_i,u_{i+1})$ the following holds $\sum_{j=i-p}^{i} N_{j,p}(u) = 1$. An important property of the basis functions is also their non-negativity, which states that a basis function will always be greater or equal to zero, no matter the degree of the curve or the number of the control points.
If a knot vector, corresponding to curve $\mathbf{C}(u)$ does not contain any internal knots, i.e. U $= 0,\ldots,0,1,\ldots,1$, and n = p, then $\mathbf{C}(u)$ simplifies to a *Bézier curve*, since it is comprised of only one knot span.
One can observe, that the control polygon of a curve forms the crudest simplification of the curve. That is, it represent the piece-wise linear approximation to which the curve simplifies to, if its order is one. The control polygon also forms a convex hull, in which the curve is contained, regardless of is degree. This can be shown based on the non-negativity and partition of unity properties of basis functions, shown previously. Specifically, if $u[u_i,u_{i+1})$, $p \leq i \leq m-p-1$, then the curve $\mathbf{C}(u)$ is the convex hull for control points $\mathbf{P}_{i-p},\ldots,\mathbf{P}_i$.

2.1.3 Definition of NURBS curves

As previously stated, NURBS are strictly-speaking not the same as B-Splines. In this section, we build on the definition of B-Splines, shown in Section 2.1.1. Based on their properties, as described in Section 2.1.2, we then discuss the differences between the two.
A NURBS curve of degree p is defined as:

$$\mathbf{C}(u) = \frac{\sum_{i=0}^{n} N_{i,p}(u)w_i}{\sum_{i=0}^{n} N_{i,p}(u)w_i}\mathbf{P}_i \qquad a \leq u \leq b, \tag{2.6}$$

where weights w_i are now newly introduced. The NURBS definition in Equation (2.6) can be reworked into a similar form to Equation (2.1) by introducing rational basis functions:

$$R_{i,p} = \frac{N_{i,p}(u)w_i}{\sum_{j=0}^{n} N_{j,p}w_j}, \tag{2.7}$$

and the NURBS curve is then defined as follow:

$$\mathbf{C}(u) = \sum_{i=0}^{n} R_{i,p}(u)\mathbf{P}_i. \tag{2.8}$$

Rational basis functions are piecewise polynomial functions, defined on certain spans of the knot vector U. Since the NURBS basis functions are based on B-Spline basis functions, the properties, described in Section 2.1.2, are valid also for rational basis functions. The effect of weights can be seen from Equations (2.7) and (2.8). They act as a scale, pulling the curve towards the control point $\mathbf{P}_i$ when the corresponding weight w_i is manipulated. One can also observe from Equation (2.7), that in case w_i = 1 for all i, the rational basis functions are equal to the B-Spline basis functions.

In Figure 2.2 a simple example of a NURBS curve is presented. It is a curve of degree $p = 2$, with a control polygon constructed from the points, shown with the black nodes. It is important to stress, that this is a conic curve, which are possible to implement only with NURBS, and not with ordinary B-Splines. Its weights, associated with each control point are:

$$w = \{1, 0.707, 1, 0.707, 1, 0.707, 1, 0.707, 1\}.$$

The knot vector is the same as the one, given in Equation 2.4.

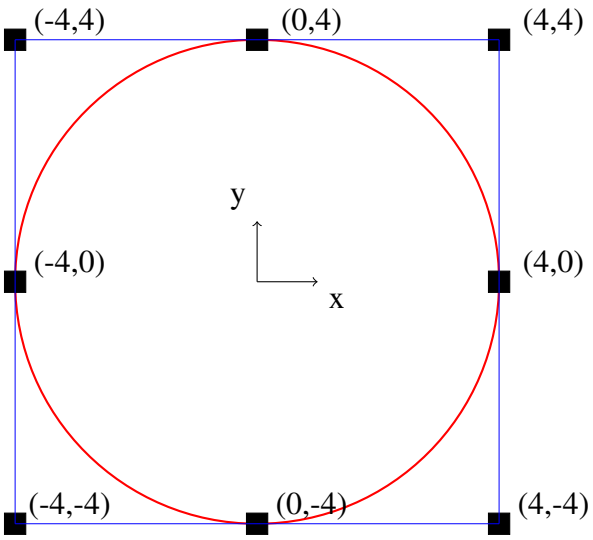


Figure 2.2: A simple example of a NURBS curve (in red), with its corresponding control points (in black) and control polygon (in blue)

Based on this, and on Equation 2.7, one can compute the rational basis functions, associated with the NURBS curve. As expected, comparing the rational basis function, shown in Figure 2.3, and B-Spline basis functions, shown in Figure 2.1, we can see that weights have a scaling effect on the basis functions, that are defined on the knot span with which they are associated with.

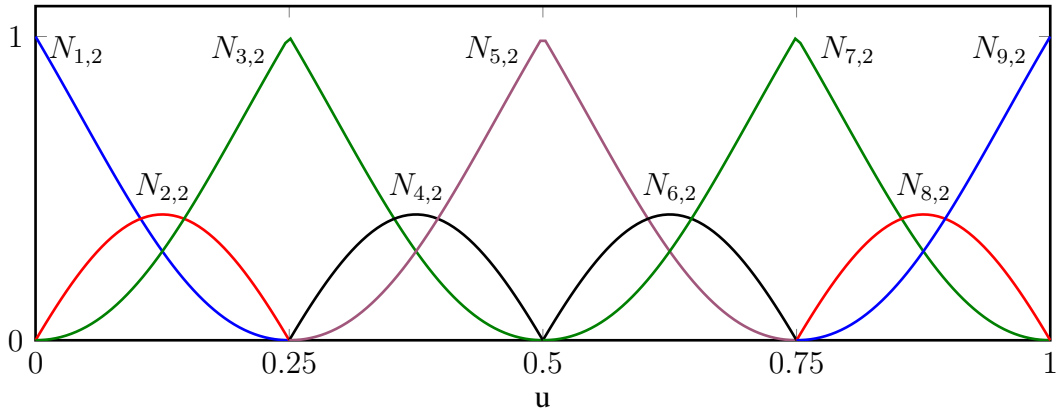


Figure 2.3: Rational basis functions of the example curve

To introduce more advanced features of NURBS curves, it is useful to describe them in homogeneous coordinates. In order to perform that, a perspective mapping H must first be introduced, which is a projection from the homogeneous space back into the original space. It is defined in [2] as:

$$\mathbf{P} = H\{\mathbf{P}^w\} = H\{x,y,w\} = \begin{cases} \left(\frac{x}{w},\frac{y}{w}\right) & \text{if } w \neq 0 \\ \text{direction}(x,y) & \text{if } w = 0. \end{cases} \tag{2.9}$$

For sets of control points and weights $\{\mathbf{P}_i\}$ and $\{w_i\}$, the homogeneous coordinates are $\mathbf{P}_i^w = \{x_i w_i, y_i w_i, w_i\}$. Note that this definition was simplified for the use in two-dimensional curves in plane. The B-Spline curve in homogeneous coordinates is then:

$$\mathbf{C}^w(u) = \sum_{i=0}^{n} N_{i,p}(u)\mathbf{P}_i^w. \tag{2.10}$$

By mapping the B-Spline back into real space, one can show that we obtain a NURBS curve:

$$\mathbf{C}(u) = H\{\mathbf{C}^w(u)\} = H\left\{\sum_{i=0}^{n} N_{i,p}(u)\mathbf{P}_i^w\right\} = \sum_{i=0}^{n} N_{i,p}(u)H\left\{\mathbf{P}_i^w\right\} = \sum_{i=0}^{n} N_{i,p}(u)\mathbf{P}_i. \tag{2.11}$$

2.1.4 NURBS Algorithm

As elaborated, implementing NURBS can be straight-forward due to their mathematical properties. In order to do that, in **function** *curvePoint* from [2], the knot vector is proportionally partitioned. A point of the NURBS curve is then calculated, corresponding to each knot value, based on definitions in Section 2.1.1. The algorithm is presented in Listing 2.1, below.

Listing 2.1: Function *curvePoint*

```
function C = curvePoint(n,degree,knots,controlPoints,u,weights)
 # INPUT: n,degree,knots,controlPoints,u,weights for the considered NURBS
 # OUTPUT: point C corresponding to the "u" parametrical coordinate

 # Applying perspective map H, defined in the NURBS Book
   for i = 1:size(controlPoints, 1)
      Pw(i,:) = controlPoints(i,:).*weights;
   end
   Pw(3,:) = weights;

   spanIndex = findspan(n, degree, u, knots); # Current knot span
   N = basisfun(spanIndex, u, degree, knots); # Value of current basis functions
   Cw = zeros(3, 1);

   for j = 1:degree + 1
      Cw = Cw + (N(j)*Pw(:, spanIndex - degree + j ));
   end
   Cw = Cw/Cw(3);
   C = Cw(1:2);
end
```

As it is seen in **function** *curvePoint*, the first variable to compute is the knot span. This means establishing the knot span, in which the parametrical coordinate of a given point is contained. A procedure is given in [2]. The pseudocode algorithm to achieve this is given below. It is based on a binary search for a match in an array. In this case, the array is the knot vector, which is also an input parameter for the algorithm, along with the parametrical coordinate, polynomial degree of the curve, and the number of the spans.

Listing 2.2: Function *FindSpan*

```
function mid = FindSpan(n,p,u,U)
 # INPUT: n,p,u,U
 # OUTPUT: i

   low = p;
   high = n + 1;
   mid = average(low,high);

   while u ! in span
      if u < U(mid)
         high = mid;
      else
         low = mid;
      mid = average(high,low);
   end
   return mid
end
```

Right afterwards, the value of the basis functions at the considered point must be computed. This can be implemented as proposed in [2], which is shown in the Listing 2.3, below. It is

based on Equation (2.3), and returns the value of each basis function at the given coordinate as output.

Listing 2.3: Algorithm BasisFuns

```
function N = BasisFuns(i,u,p,U)
 # INPUT: i, u, p, U
 # OUTPUT: N

   for j = 0:p
      left(j) = u - U(i + 1 - j);
      right(j) = U(i + j) - u;
      saved = 0;
      r = 0
      for r = 0:j + 1
         temp = N(r)/(right(r + 1) + left(j - r));
         N(r) = saved + right(r + 1)*temp;
         saved = left(j - r)*temp;
      end
      N(j) = saved;
   end
   return N
end
```

2.1.5 Knot Insertion

As noted earlier, knot insertion is usually performed on the NURBS defined in homogeneous space, which is shown in Equation (2.10). Here we follow the definition, as described in [2]. The NURBS in homogeneous coordinates $\mathbf{C}^w(u)$ is defined over a knot vector of the form $\mathrm{U} = \{u_0, \ldots, u_m\}$. Now, a new knot span $\bar{u} = [u_k, u_{k+1})$ can be defined, and inserted into U to create a new knot vector $\bar{U} = \{\bar{u_0} = u_0, \ldots, \bar{u_k} = u_k, \bar{u_{k+1}} = \bar{u}, \bar{u_{k+2}} = u_{k+1}, \ldots, \bar{u_{m+1}} = u_m\}$. The vector spaces, that include the knot vectors are denoted as $\mathcal{V}_U$ and $\mathcal{V}_{\bar{U}}$. Because the original knot vector has been modified, it follows that $\mathcal{V}_U \subset \mathcal{V}_{\bar{U}}$, and consequently $\mathbf{C}^w(u)$ can be written in the parametrical space of $\bar{U}$ as:

$$\mathbf{C}^w(u) = \sum_{i=0}^{n+1} \bar{N}_{i,p} \mathbf{Q}_i^w, \tag{2.12}$$

where $\bar{N}_{i,p}$ are basis functions, defined over the knot vector $\bar{U}$, and $\mathbf{Q}_i^w$ are the corresponding control points. One can observe, that knot insertion is merely a process of finding new control points that correspond to the new knot vector, based on the old one. Knot insertion can thus be seen as a change of vector space, where basis functions, which construct a NURBS curve, along with their corresponding control polygon are changed, whereas the curve itself physically does not change.
One can indeed produce the formula for finding the new control polygon from this fact. Here we skip the derivation, which is shown in [2], and show only the formula for obtaining the new

control points as a linear combination of old ones:

$$\mathbf{Q_i}^w = \alpha_i \mathbf{P}_i^w + (1 - \alpha_i)\mathbf{P}_i^w, \tag{2.13}$$

where α_i varies based on the knot span:

$$\alpha_i = \begin{cases} 1 & i \leq k - p \\ \dfrac{\bar{u} - u_i}{u_{i+p} - u_i} & k - p + 1 \leq i \leq k \\ 0 & i \geq k + 1 \end{cases} \tag{2.14}$$

of the basis functions is non-zero at that knot.

As a NURBS curve is split into several curves, it is possible to redefine them in order to use them later on more efficiently. Such a process is performed by repeatedly inserting a knot into the knot vector, until the multiplicity of the knot $k = p$, and thus a C^0 continuity at the parametrical coordinate of the knot is achieved. It is again important to note, that the curve itself is physically not split, only the basis functions are, while the control polygon is appropriately modified to suit them.

Listing 2.4: Algorithm *CurveKnotIns*

```
function Qw = CurveKnotIns(np,p,UP,Pw,u,k,s,r)
 # INPUT: np, p, UP, Pw, u, k, s, r
 # OUTPUT: nq, UQ, Qw
   mp = np + p + 1;
   nq = np + r;

   # Load new knot vector
   for i = 1:k   UQ(i) = UP(i);  end
   for ii = 1:r  UQ(k + 1) = u;  end
   for j = k + 1:mp  UQ(j + r) = UP(i); end

   # Save CP that do not change
   for jj = 0:k - p Qw(jj) = Pw(jj); end
   for ii = k - s:np  Qw(ii+r) = Pw(i); end
   for ii =0:p - s  Rw(ii) = Pw(k - p + 1); end

   # Knot is inserted r-times
   for jj = 1:r
      L = k - p + j;
   end
   for i = 0:p-j-s
      alpha = (u - UP(L + i))/(UP(i + k + 1) - UP(L + i));
      Rw(i) = alpha*Rw(i+1) + (1 - alpha)*Rw(i);
      Qw(L) = Rw(1);
      Qw(k + r - jj - s) = Rw(p - j - s);
   end
   for i = L:k - s
      Qw(i) = Rw(i - L);
   end
return nq, UQ, Qw
end
```

In Listing 2.4, above, the algorithm, proposed by [2] for performing knot insertion, is shown. As input, the parameters of the NURBS curve under consideration – degree of the curve p, control points $\mathbf{P}$, knot vector U, new knot to be inserted u, and its multiplicity r. The knot is r-times inserted into the knot vector, where for each insertion, new control points are calculated, based on Equation (2.13). Finally, the new control points, and new knot vector are returned.

2.2 Quadtree Decomposition

In this section, quadtree decomposition and its associated data strucuture shall be presented, along with some properties and advantages that it offers.
When simulating the behaviour of heterogeneous materials under the influence of some physical phenomena, where one has to deal with several different materials being enclosed in the domain, respecting the interfaces between the materials can be challenging. If one chooses to use the Finite Element Method to perform such an analysis, the domain must be partitioned into elements. These then form a discrete mesh, which heavily influences the accuracy of computed results. As a general rule, the finer the mesh is, more accurate the approximations will be. However, a finer meshes also means a greater computational cost of the calculation. It is in such cases pragmatic to refine the mesh only where needed, i.e. at interfaces of different materials, while keeping the mesh at other locations of the domain coarser. One of the mesh generation methods that is non-uniform and can fulfill the above-stated rules is the quadtree decomposition. It is based on quadtree data structures, where a quadtree is a rooted tree in which every node has four children, and each represents a quadrant of the parent node, hence the name.
Figure 2.4 shows an example of a quadtree decomposition of a square root (domain), with its corresponding tree structure. Also shown is the naming convention of the children quadrants.

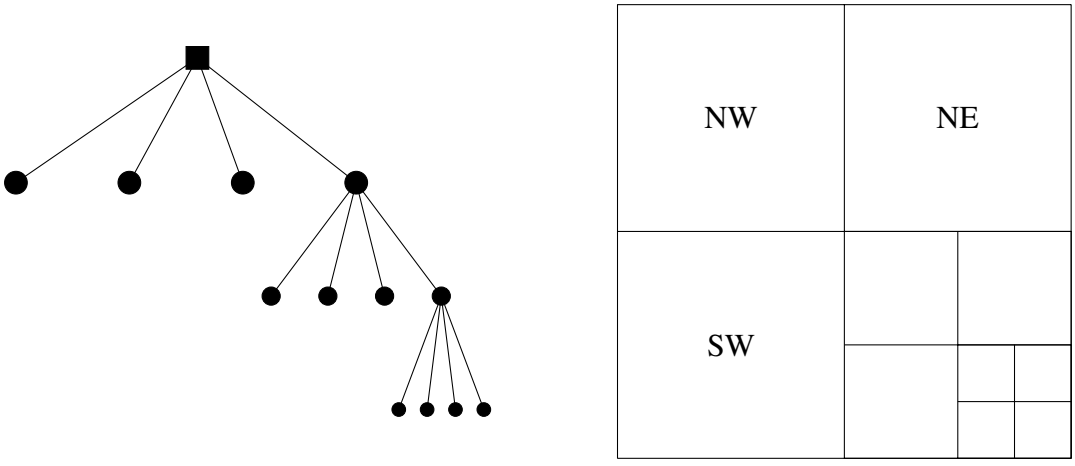


Figure 2.4: Quadtree decomposition of a square domain and its corresponding tree structure

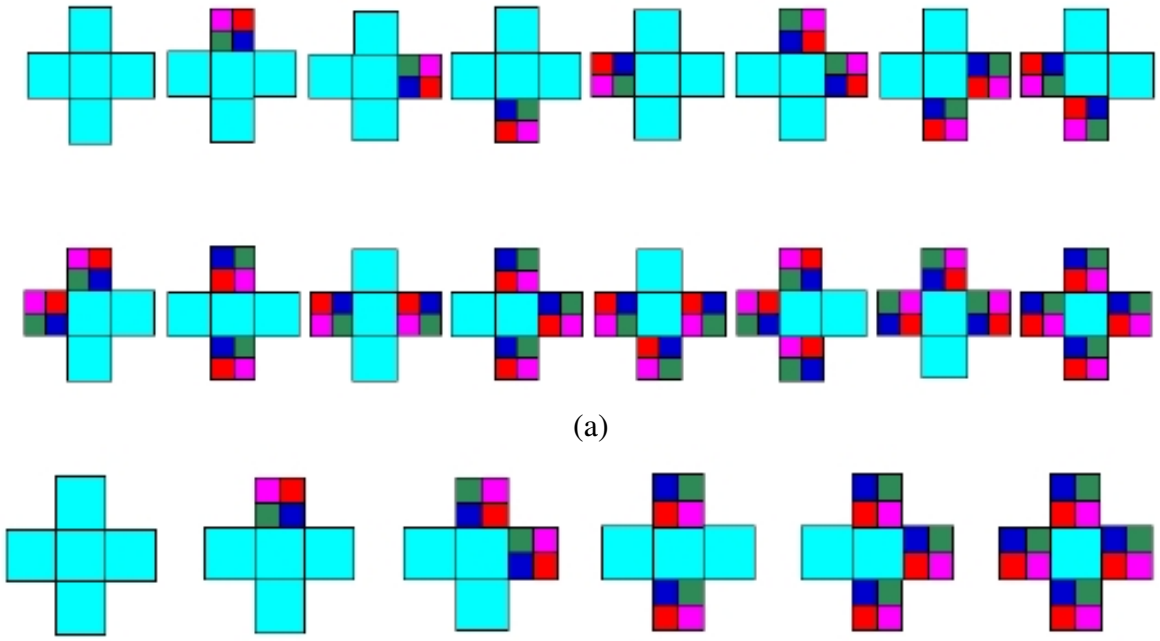
2.2.1 Decomposition criteria

Seed points The decomposition can be performed based on a variety of criteria. Here, quadtrees will be based on point data sets. That is, if more than one of some points are present in a certain quadrant on a level of decomposition, then the quadrant in question is split into four children quadrants.
If the point set consists of only one point, then the quadtree is constructed from only one leaf (node at maximum decomposition level), and only one square. In the case that point set P consists of more than one point, then we note σ_{NE}, σ_{NW}, σ_{SE}, σ_{SW} as the four quadrants of the square σ. We then define them by using $x_{mid} = (x_\sigma + x_\sigma^{'})/2$ and $y_{mid} = (y_\sigma + y_\sigma^{'})/2$, according to [3]:

$$
\begin{aligned}
P_{NE} &= \{p \in P : p_x > x_{mid} \text{ and } p_y > y_{mid}\},\\
P_{NW} &= \{p \in P : p_x \leq x_{mid} \text{ and } p_y > y_{mid}\},\\
P_{SW} &= \{p \in P : p_x \leq x_{mid} \text{ and } p_y \leq y_{mid}\},\\
P_{SE} &= \{p \in P : p_x > x_{mid} \text{ and } p_y \leq y_{mid}\}.
\end{aligned}
$$

Thus, point sets for squares, which correspond to the respective children of square σ, and represent the root of the quadtree have been defined. This is then one of the stop criteria for what is otherwise a recursive process of root decomposition.

2:1 Rule Another criteria, which will be implemented, is one based on common practice in order to balance the decomposition. This is the so-called 2:1 rule, which states that, if two quadrants share a side, then their level of decomposition can be at most one level different. Here it is important to note, that the ratio rule is not obligatory per-se, rather it is a convention, that is often applied in literature in order to reduce of possible quad combinations of the resulting mesh. This then simplifies the procedure considerably. It is illustrated in Figure 2.5.

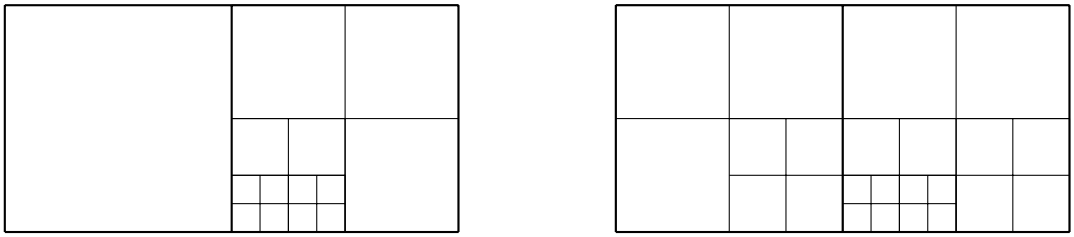


(a)

(b)

Figure 2.5: Comparison between (a) sixteen possible combinations of quad configurations, and (b) six combinations possible removing symmetrical cases [4]

In Figure 2.6, a simple example is presented in order to illustrate the effect of applying the 2:1 rule. Note that this figure does not show a FE mesh, but rather a root fully decomposed into a non-uniform grid. If we were to create a discretization from this decomposition, one can observe, that Figure 2.6(b), which implements 2:1 rule, would produce a more balanced mesh.



(a) (b)

Figure 2.6: Comparison between domains: (a) not conforming, (b) conforming to the 2:1 rule

Quad-balancing is often implemented using neighbor-finding algorithms. Basically, this involves finding node ν', based on the node ν, so that $\sigma(\nu')$ neighbors $\sigma(\nu)$ in the chosen direction. An algorithm for searching such neighbor nodes is proposed in [3]. Based on a NW node ν in question, and quadtree data structure τ, it recursively finds the north neighbor μ of the parent of ν. The north neighbor of ν is going to be a child of of μ, if the latter is an internal node - but, if it is a leaf, then the neighbor being searched for is in fact μ.

Listing 2.5: Function NorthNeighbor

```
function m = NorthNeighbor(v)
 # INPUT: Node v in quadtree tau
 # OUTPUT: Deepest node m so that its depth is at most
 #  v such that quad(m) is a north neighbor
 #  of quad(v) or null if there is no such node.
   if v = root(tau)
      return [];
   if m is SW-child of parent(v)
      return NW-child of parent(v);
   if m = SE-child of parent(v)
      return NE-child of parent($\nu$);
   m = NorthNeighbor(parent(v),tau)
   if n = [] || m is a leaf
      return m
   else if m = NW-child of parent(v)
      return SW-child of parent(v)
   else return SE-child of v
end
```

Based on this, [3] also proposes an algorithm for quadtree balancing with the algorithm, shown below in Listing 2.6.

Listing 2.6: Function BalanceQuadTree

```
function tau1 = AlgorithmBalanceQuadTree(tau)
 # INPUT: A quadtree
 # OUTPUT: A balanced quadtree, based on tau
   while L != []
      remove a leaf v from L
      if quad(v) has to be split
          Make v into an internal node with four children,
          which are leaves, that correspond to four quadrants of quad(v).
          If v stores a point, then store the point in the correct new leaf instead
          Insert the four new leaves into L
          Check quad(v) regarding neighbors that need to be split, and, when neccessary,
            insert them into L
   return tau1
end
```

Two Intersections Per Quad After both of the previously described decomposition criteria, another state must be considered. It is possible, when dealing with complex interfaces, that the quadrant, corresponding to a leave, contains more than two interface points.
In such cases it is therefore useful, that additional decomposition is performed, until one is left only with leaves containing at most two different intersection points. Verifying this can be connected to calculation of intersection. In such cases having more than two intersections is undesirable. It should be stressed, that this decomposition criteria simplifies the materials identification in a quad that contains a segment of the interface and hence different types of materials.

Star-Shape Condition If the one-interface-per-quad condition is considered, then each leaf of the quadtree that is not empty, has a piece of the original NURBS curve. If we consider the edges of a quad leaf element as boundaries of a domain, the piece of the interface inside the leaf divides the domain into two subdomains.
These subdomains must be star-shaped if we wish to use the Scaled-Boundary Finite Element Method [5]. Due to this important fact, the following Section 2.3 is dedicated to this criteria.

2.3 Star-Shaped Polygons

Following the idea of the Scaled-Boundary Finite Element Method (SBFEM), a star shaped element geometry is needed for considered subdomains [5]. This allows for the correct positioning of the scaling center inside each subdomain, and thus is an additional condition, which must be satisfied when applying the quadtree decomposition algorithm.

The position of the scaling center is chosen, so that the total boundary of the subdomain is visible. This is possible only at a certain set of points inside the subdomain, called the kernel. A subdomain that has a kernel is considered star-shaped. Convex polygons are by definition star-shaped, and a convex polygon coincides with its own kernel. For concave polygons, an algorithm searches for the kernel. If such is found, the subdomain is star-shaped, hence no further decomposition is needed for this leaf element. If a kernel cannot be found, meaning the subdomain is not star-shaped, further decompositions must be performed on the given element until the two resulting subdomains become star-shaped.

An algorithm for finding the kernel of a simple polygon is proposed by [6]. The following Listing 2.7 represents the concept of this algorithm.

Listing 2.7: Function *starShapedCheck*

```
function [K] = starShapedCheck(vertices)
 # INPUT: vertices of the polygon K
 # OUTPUT: kernel of the polygon K if star-shaped, otherwise K is empty

   # Check global orientation of polygon
   # -----------------------------------
   # -1 = clockwise, 1 = counter-clockwise
   orientationPolygon = evaluateOrientation(vertices);

   for i = 1:length(vertices)
      # Compare global polygon orientation to local vertex orientation
      # to identify concave vertices. In case orientations don't match
      # vertex is concave.
      store concave vertices and their coordinates
   end
   if concaveVertices = [];
      return K = [];
   end

   # Initialize
   # ----------
   K = vertices;
   for v = 1:length(vertices)
      elongate edges of the vertex v until it intersects with polygon

      calculate intersection points with the polygon

      insert intersection points into K

      delete vertices outside of resulting kernel from K
   end
   # Make sure K is not a line object
   if length(K) < 3
      return K is empty;
   else
      return K;
end
```

The algorithm first identifies all the concave vertices. If none are found, the polygon is concave and coincides with its own kernel. Otherwise, at least one convex vertex is present, and a kernel must be found. In that case, the coordinates of all vertices are written in a matrix K. For each convex vertex, the following operations are performed: First, an intersection of its elongated edges with the rest of the polygon is found and the coordinates of the intersection points are added to the matrix K. Then vertices on the outside of the elongated edges are removed from the matrix K, reducing the area of the original polygon. These steps are then repeated for all convex vertices, which reduces the size of K to the kernel of the polygon.

3 Implementation

In this section, an implementation task, defined in Section 1, is presented. A flowchart of the developed routine is given below, in Figure 3. After giving initial inputs, which are the parameters that represent the interface between two different materials, a NURBS curve that describes the interface is computed. The first condition to be satisfied is, that each quad element must have a maximum of one control point of the original NURBS curve. If that is not the case, a quadtree decomposition is performed again, until this criterion is reached. Following each decomposition, if any of the resulting children contains a segment of the interface, curve splitting is performed. This consists on obtaining the definition of the NURBS segment that is contained in the children quad, split from the rest of the NURBS curve. Next step is to check whether the resulting polygons are star-shaped, followed by the control of the quadtree balance. If either one of those checks fails, a quadtree decomposition is performed again and the whole procedure, starting at the decomposition, must be repeated.

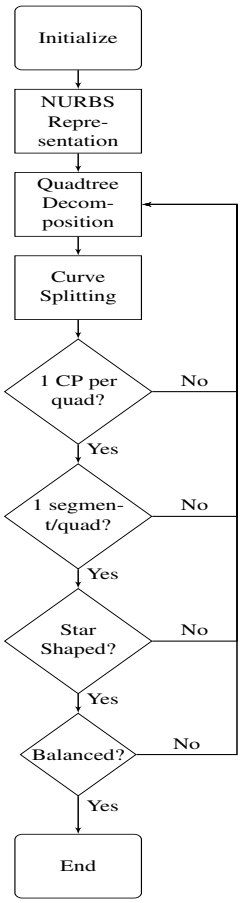


Figure 3.1: Illustration of the workflow for the implementation

3.1 NURBS Representation

As explained in Section 2.1, in order to represent a NURBS curve in geometrical space, one requires information regarding the polynomial degree of the curve in question, its knot vector, control points, and weights corresponding to said control points. In Listing 3.1, below, we present **function** *calculateNURBS*, which takes the mentioned parameters as input, and returns a NURBS curve as a matrix of x- and y-coordinates, along with its parametrical coordinates.

Listing 3.1: Function *calculateNURBS* -

```
# INPUT: Parameters, used in NURBS curve calculation
# OUTPUT: A NURBS matrix, where:
#  1st row: x coordinates in physical space,
#  2nd row: y coordinates in physical space,
#  3rd row: parametrical coordinates.

  counter = 1 ;# Initialize counter
  n = length(knots) - degree - 2; # Number of knot spans

  for i = knots(1):0.01:knots(end)
     # Loop over parametrical space defined by the knot vector
     # Obtain each coordinate of the NURBS curve and store it
     [f] = curvePoint(n, degree, knots, controlPoints, i, weights);

     NURBS(1:3, counter) = [f(1);f(2);i];
     counter = counter + 1;
  end
```

In Figure 3.2 we present a NURBS curve, which represents an interface boundary. We will use it to provide illustrations of the implemented functionalities. Note, that the control polygon is shown with a dotted line.

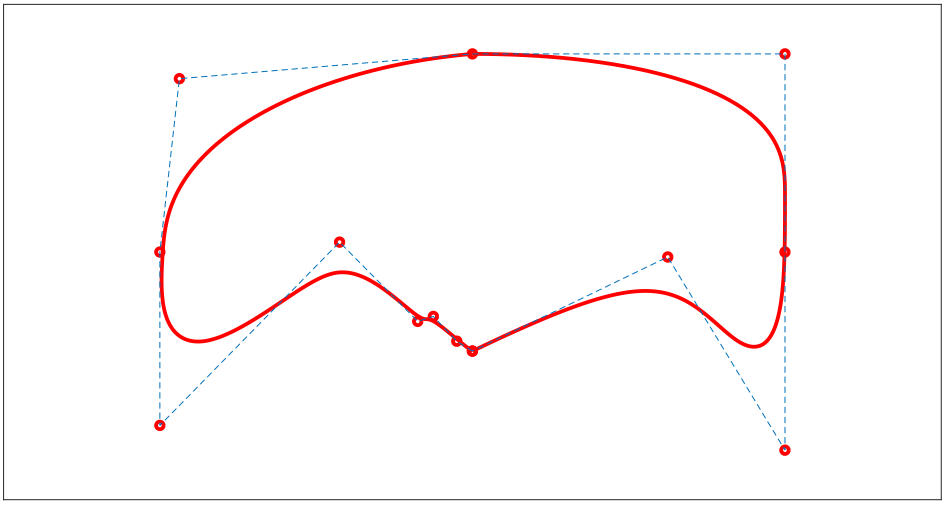


Figure 3.2: Example of a complex NURBS curve

3.2 Quadtree Decomposition of the Domain

Following interface-representation with NURBS, one can begin performing quadtree decomposition of the domain. This is performed using the **function** *decompose*, shown below, in Listing 3.2. This is, as mentioned, a recursive function, and firstly divides a given quad into four children. Afterwards, the function to split the NURBS curve in each child quad is called. Due to the need for having only one curve segment in each quad, this must be verified. If there is more than one segment or control point in quad, then an additional decomposition needs to be performed.

Listing 3.2: Function *decompose*

```
function [Quadtree] = decompose(Quadtree,Quad,NURBS, controlPoints,
            knots, weights, degree, l, k,pos_aux, Q_aux, Boundary)
 # INPUT:
 # Quadtree data
 # Quad: geometrical definition of the quad that is about to be splitted
 # Definition of the NURBS
 # l: - 1 if quad NW,
 # - 2 if quad SW,
 # - 3 if quad NE,
 # - 4 if quad SE. Initializate at 1
 # k: level of decomposition
 # OUTPUT:
 # Quadtree data after performing decomposition and eventual curve splitting

   # Split the parent quad into four child quads
   children = splitQuad(Quad);

   for i = 1:4
      # Loop over the 4 children. First the eventual NURBS segment contained
      #in the given children is split. Then check if there is more than
      # one seed point or two segments of the curve in the quad. If one of
      # those two conditions is true, decompose the given quad

      # Selecting given quad
      Current = select(children);

      # Call the splitting function. Get the tree data structure after
      # performing the splitting at the given quad and an auxiliary array
      [Q_aux, Quadtree] = splitting(Quadtree, Q_aux, Current, NURBS, ...
      controlPoints, knots, weights, degree,l, k, i, pos_aux);

      # Obtains number of NURBS segments in given quad
      numInterPoints = checkNumCurvesInQuad(Current, NURBS, degree, knots,...
      controlPoints, weights);

      # Obtains number of seed points in given quad
      nPointsCurrent = checkQuad(Current(:,1), Current(:,2),
                     Boundary(:,2), controlPoints);
      # If there are more than two segments or seed points, decompose again
      if nPointsCurrent > 1 || numInterPoints > 2
         [Quadtree] = decompose(Quadtree, Current, NURBS, controlPoints, knots,...
         weights, degree, l, k, pos_aux, Q_aux, Boundary);
      end
   end
end
```

The procedure of a quadtree decomposition is shown in Figure 3.3. The geometry corresponds to the one, shown in Figure 3.2. Figure 3.3(a) shows the first level of decomposition, Figure 3.3(b) the second, Figure 3.3(c) the third, while Figure 3.3(d) illustrates the last stage of the decomposition.

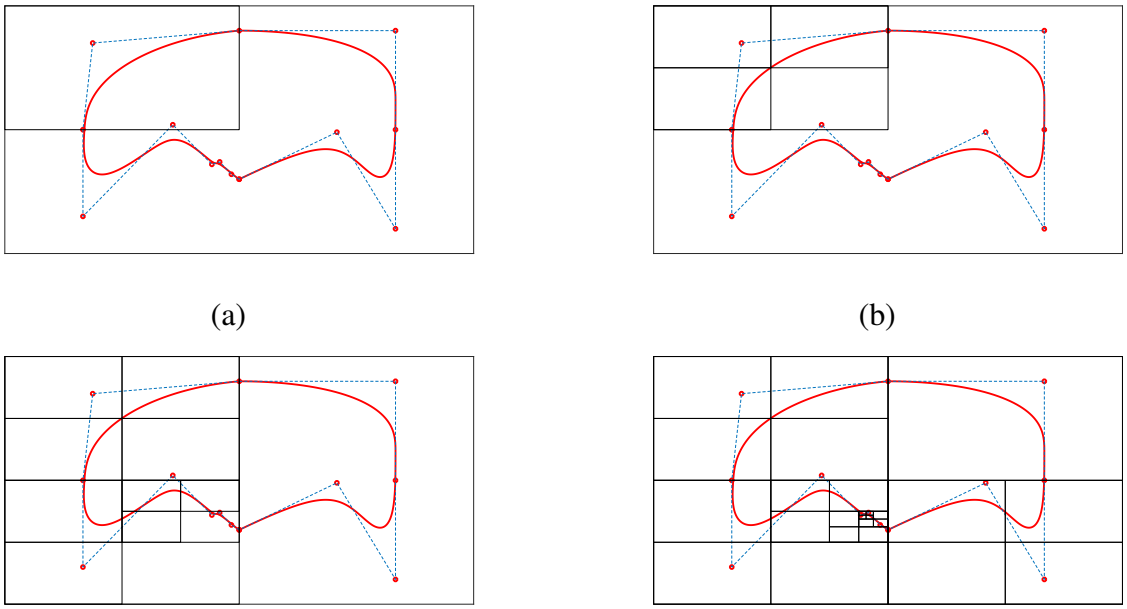


Figure 3.3: Illustration of a root decomposition in sequence (a) – (d)

3.3 Curve Splitting

Curve splitting is performed interchangeably with quadtree decomposition. **Function** *splitting* is introduced to perform this, and is shown in Listing 3.3. It first calls **function** *Inter* and obtains the intersection points between the NURBS and the considered quadrant, if they may exist. After intersection has been determined, knot insertion of its parametrical coordinates is performed, until C^0 continuity is obtained, as described in Section 2.1.2. This is done repetitiously by **function** *CurveKnotIns*, as depicted in Section 2.1.5. With it, we obtain the NURBS description of the curve segment, that will we be eventually enveloped in a certain quad. Afterwards, the curve segment, pointers and auxiliary variables are stored in the datatree at the node, assigned to the given quad.

The intersection points between the segments of any given quad and the NURBS curve needs to be obtain in first place. This is done with the function Inter, shown in Listing 3.4, below. If such an intersection exists, the control polygon will gernerally also intersect the quad.

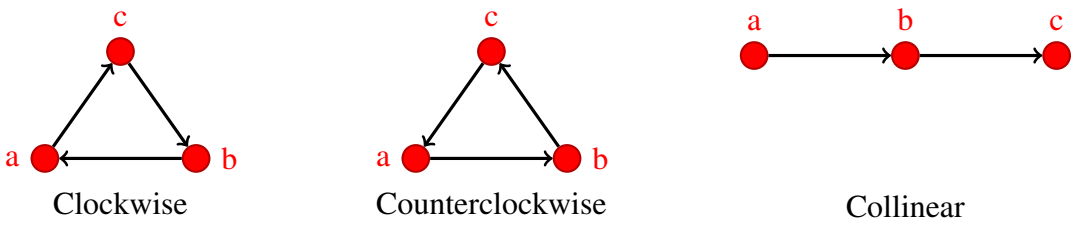


Figure 3.4: Example of orientation of polygons

Figure 3.5: Intersection conditions of two segments

The **function** *Inter* considers an edge of a quad and each segment of the control polygon, once at a time. It computes the orientation between the two segments in triplets, as described above. If the intersection condition is fulfilled, then it continues making a first approximation of the intersection. This is obtained by the **function** *findinterval*. It computes the section of the curve where the control points that define the intersecting segment have influence, i.e. where their associated basis functions do not vanish. Afterwards, it obtains the closest point - of the ones which conforms this section of the curve - to the intersection. At the end a modified Steffensen's method is applied. This is a modified Newton-Rhapson method for obtaining a numerical solution with quadratic convergence and without using derivatives in the formulation. The function where the root is searched defines the orientation between the last approximated point and the intersecting edge of the quad, normalized by the length of the quad. If this function takes a value of 0, then the quad's segment and the obtained point would be collinear. This means that this is the intersecting point of the NURBS and the considered edge of the quadrant.

It is important to notice that determining the intersection between NURBS and a segment is not a trivial task. There is extent literature regarding this problematic, mostly at the field of animation and graphic design, focused on intersections between two NURBS functions [7], [8], [9].

Listing 3.3: Function *splitting*

```
function [Quadtree] = splitting(Quadtree, Quad, controlPoints,...
knots, weights, degree, auxliar_vars)
 # INPUT:
 # Quadtree data
 # Geometrical definition of the considered quadrant
 # Definition of the NURBS
 # Auxiliary variables previously defined
 # OUTPUT:
 # Quadtree data structure containing the splitted
 # segment of the NURBS

   # Obtain possible intersections
   # -----------------------------
   [P,U] = Inter(Quad,degree,knots,controlPoints,weights);
   # Intersection above horizontal
   [P,U] = Inter(Quad,degree,knots,controlPoints,weights);
   # Intersection left vertical
   [P,U] = Inter(Quad,degree,knots,controlPoints,weights);
   # Intersection right vertical
   [P,U] = Inter(Quad,degree,knots,controlPoints,weights);

   # Knot insertion, based on intersection
   # -------------------------------------
   newKnots = U;
   knots_new = knots;
   for j = 1:length(newKnots)
      newKnot = newKnots(j);
      for ij = 1:degree # Insert knot untill C^0 continuity
         [knots_new, controlPoints, weights] = CurveKnotIns(degree,...
         controlPoints, knots_new, weights, newKnot);
      end
   end
   # Store in datatree
   # -----------------
   [Quadtree] = savetree(Quadtree, Quad, degree, controlPoints, knots_new,...
   weights, P, auxliar_vars);
end
```

Listing 3.4: Function *Inter*

```
function [Pint,U] = Inter(A,B,degree,knots,controlPoints,weights)
 # INPUT:
 # x1,y1,x2,y2 - rectangle coordinates
 # NURBS curve
 # OUTPUT:
 # Pint: physical coordinates of the intersection point(empty if none)
 # U: parametrical coordinates of the intersection point(empty if none)

   # Initialisation
   # ---------------
   # Determine whether there is intersection between control polygon segment
   # and quad edge
   for i = 1:size(controlPoints,2) # Loop over the control points
      # Each segment of the control polygon is defined by two consecutive
      # control points.
      P = controlPoints(i); Q = controlPoints(i);

      # Checks 4 orientations between Pi,Qi,A,B
      OI, OII, OIII, OIV = orientation(A, B, P);
      # If the condition is fulfilled there is a intersection. Obtain the
      # indexes of the control point array that defines the intersecting
      # segement and the number of intersections.
      if OI ~= OII && OIII ~= OIV  index= [i i+1]; end
   end

   # First aproximation of the solution
   # ----------------------------------
      for i = 1:num_intersected_segments
         # Loop over the segments of the control polygon that intersects the
         # quad's edge
         # findinterval scans the part of the NURBS controlled by the control
         # points that define the intersected segment. Obtains the first
         # approximation of the solution in parametrical coordinates as    output
         [aprox] = findinterval( cond3,idx2, index, n,degree,knots,...
      end

   # Solution based on Steffensen's method
   # -------------------------------------
      for i=1:num_approximated_solutions # Loop over the approximate solutions
         a = aprox(i);
         for ii=1:10
            [f] =

            (n,degree,knots,controlPoints,a,weights);
            # lfunc function to be minimized
            O1 = lfunc(f,A,B);
            if err < tol break; end
            [g] = curvePoint(n,degree,knots,controlPoints,a+hh,weights);
            [O3] = lfunc(g(1),g(2),x1,y1,x2,y2);
            a = a - O1/((O3-O1)/(hh));
         end
      end
end
```

This process is exemplified with the NURBS curve described at 3.1. In this case the splitting at the bottom left quadrant is performed. **Function** *splitting* first searches for intersection points between each of the edges of the quadrant and the NURBS curve. In this case two intersection points are computed, marked in black at Figure 3.7. Right afterwards the parametrical coordinates associated with those two intersection points are inserted in the knot vector. This is done until a C^0 continuity of the basis functions at those parametrical coordinates is achieved. Once the new control points and weights are obtained, the segment of the NURBS contained in the quad is split from the rest of the curve, as it can be observed at Figure 3.7.

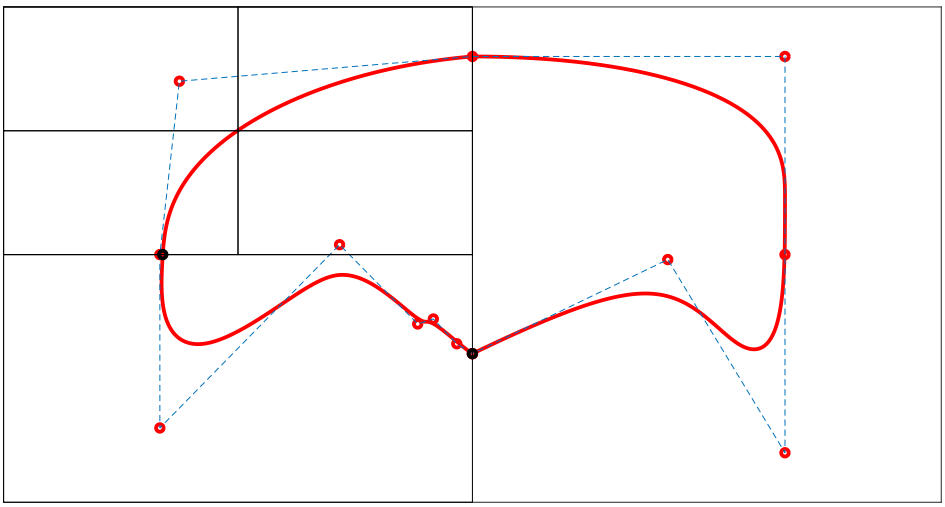


Figure 3.6: Illustration of intersection-finding procedure, intersections denoted with black circles

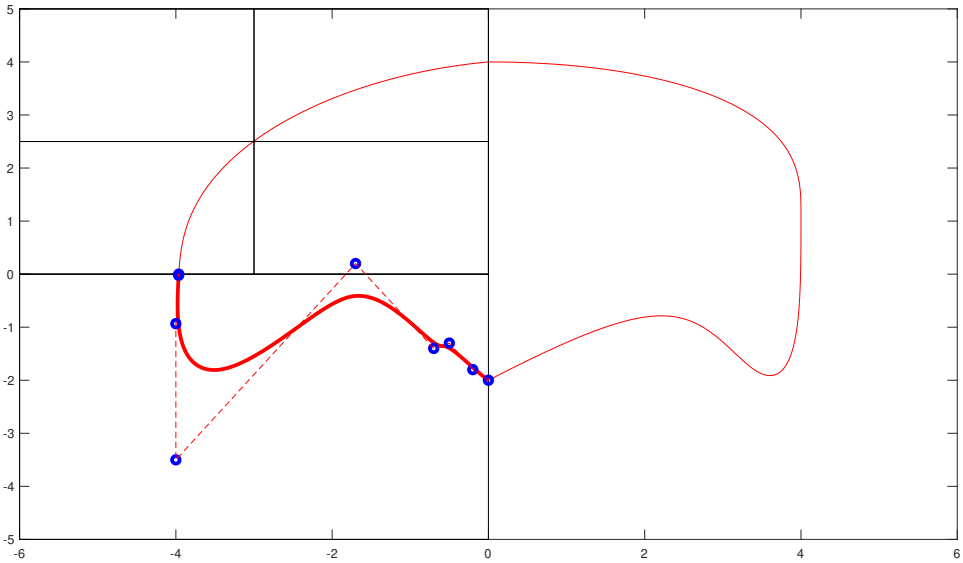


Figure 3.7: Example of a split curve (shown in red)

3.4 Tree Data Structure

The data, generated by the program, must be stored in a proper manner. For this purpose, a tree data structure is employed, which can mirror the structure of the resulting decomposition. This allows us to navigate upwards and downwards along the levels of decomposition, by storing pointers to parents and children of nodes, to which each quad corresponds.

Row number	Contains	Example	Empty if
1	Quad name	'Quad2 2 1 2'	–
2	Direction	[2,2,1,2]	–
3	Inter. horizontal	[3.999;-1.834e-15]	Only 1 subdomain
4	Inter. vertical	[3;-1.279]	Only 1 subdomain
5	Inter. parametrical	[0.819,0.925]	Only 1 subdomain
6	Degree	3	It has 4 children
7	Control Points	[x; y coordinates]	It has 4 children
8	Knot vector	[0 0 0 0 0.5.1 1.1]	It has 4 children
9	Weights	[1 1 0.7 1]	It has 4 children
10	Quad definition	–	–
11	Pointers to children	[3 8 17 34]	It has no children
12	Kernel 1	[x; y coordinates]	Not star-shaped
13	Kernel 2	[x; y coordinates]	Not star-shaped

Table 3.1: Example of an array for storing tree data structure

In Listing 3.5, below, **function** *savetree* prepares the information that will be stored in the tree data structure. After splitting, the function for the description of the NURBS, contained in the quad is stored in a node of the tree data structure, with its respectives pointers and auxiliary data. The code from [10] is used for generating a tree data, by introducing the quadtree as a MATLAB class.

The data is stored after each decomposition of the quadtree. The function only stores the information of the NURBS contained in the quad, if it contains two subdomains. When a quad has four children this information is erased, so that there is no information duplicity. While checking the star shape condition of the quads, the resulting kernels of the element will be stored. Each node contains a cell with thirteen rows. The pointers to the parents for each node are stored in an additional array. An example of such an array is shown in Table 3.1.

The direction of a given quad, representing a node in the tree data structure is obtained as follows: first, the value 11 is assigned to a NW quad, 21 to a SW quad, 12 to a NE quad, and 22 to a SE quad. The position of a quad in a given quadtree is composed by the position of each ancestor with respect to the considered father. For example, the colored quad at Figure 3.8 has the direction 22 11. Its father has a NE position with respect to his own father, and thus the first direction is 22. The considered quad has a NW position towards its father, i.e the complete location of this quadrant is 22 11.

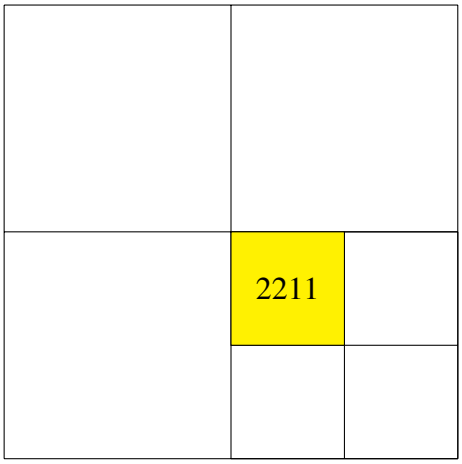


Figure 3.8: Example of system in place for quad referencing

Listing 3.5: Function *saveTree*

```
function [Quadtree] = savetree(Quadtree, Quad, degree, newKnots, ...
controlPoints, knots, weights,auxliar_vars)
 # INPUT:
 # Quadtree data
 # Quad: geometrical definition of the considered quad
 # Definition of the NURBS
 # newKnots: parametrical coordinates of the previously inserted knots
 # Auxiliary variables previously defined
 # OUTPUT:
 # Quadtree after storing new information

 # Initialisation
 # --------------

   for j = 1:length(newKnots)/2
      # Loop over the inserted knots and select sections of the NURBS,
      # contained in the quad. First, the part of the knot vector,
      # control points and weigths that define the NURBS at quad have
      # to be selected
      # Vectors controlPoints_store, knots_store and weights_store contain
      # this information

      k0 = find(first index of knots == newKnots(j));
      k1 = find(first index of knots == newKnots(j+1));
      controlPoints_store = controlPoints(1:2,(k0-1):(k1-degree));
      knots_store = [knots_new(k0) knots_new(k0:k1) knots_new(k1)];
      weights_store = weights((k0-1):(k1-degree));
   end

   # Storing information at tree's new node and attaching it to father. Data
   # defines the information to be stored.
   data = {Name; direction; intersections P; intersections u; degree;...
   controlPoints_store; knots_store; weights_store; Quad};

   # A new node on the tree data structure is created, data, needed to store is
   # given as input, along with the direction of the father. Output is the
   # Quadtree after adding node and direction of the new node at the tree
   [Quadtree, node] = Quadtree.addnode(Parent, data);

   # Adding pointer 'node' to father's new child
   Quadtree = Quadtree.set(Parent, node);

   # If a given node has already four children, NURBS information doesn't need
   # to be stored there, since it is also stored in the children
   if length(Pointers_child) == 4
      Erase NURBS information
   end
end
```

3.5 Star-shaped Polygons

In Section 2.3, a kernel searching algorithm was introduced. The current section deals with its applications on the resulting subdomains inside a leaf element from the quadtree grid. An important condition for the use of the routine is the non-self-intersecting of the polygons, that represent the subdomains. The two resulting subdomains in each leaf element are represented by the boundaries of the leaf and the piece of the original NURBS curve inside the element. For each of the subdomains, a kernel must be found.

For achieving the task, two function are proposed. *Algorithm 1* uses the control polygon of the NURBS curve to divide the element into two subdomains. This is only possible, when all the control points are located inside the quadtree leaf boundaries, hence the subdomains are non-self-intersecting polygons. The algorithm allows a quick check regarding the shape of the subdomains and does not require a polygonization of the curve. Most of the times this is adequate approach, as the control polygon approximates the curve with sufficient accuracy. This is due to the repeated decomposition and curve splitting. Most of the times *Algorithm 1* will fail by manipulating the input data and forcing this to happen. For general use, however, it is not expected that another solution will be needed.

The following lines describe how the two subdomains are obtained when quad and control polygons are given. First, the vertices coordinates of the quad are stored in an array, named polygon, in counter-clockwise direction. The NURBS curve in the quad, that represents the domain, is calculated. The control points of the NURBS curve are stored in array, named points. Here it must be mentioned, that the direction of control points is no strictly specified and may vary from quad to quad. Next, the **function** *starSplitPolygons* is called, which first finds the intersection points of the quad and the control polygon. These are then inserted into an array *polygon* between the corresponding vertices, while respecting the direction of rotation. The next step is to split *polygon* into *polygon1*, and *polygon2*, where the intersection points have been previously added, while at the same time adding the control points to the two resulting subdomains. The non-trivial part is, that the arrangement of the control points might have to be inverted, depending on the orientation of the control polygon and the subdomains. Finally, polygons 1 and 2 are returned and the kernel searching algorithm can be run. Listing 3.6, below, shows a pseudocode of the **function** *starSplitPolygons*.

Listing 3.6: Function *starSplitPolygons*

```
function [polygon1,polygon2] = starSplitPolygons(Quad_current,points,Int)
   # Input: Quad_current (domain to be split),
   # points (the splitting interface), Int (intersection points)
   # Output: polygon1, polygon2

   # INITIALIZE
   # ----------

   polygon = Quad_current;

   # Code to insert intersection points in quad polygon
   for l = 1:2 # Loop through intersection points
      for i = 1:4 # Loop through quad vertices
         find where to insert the intersection points
         insert intersection points in the quad polygon
      end
   end

   # The locations to cut the quad polygon when inserting the vertices of
   # the control polygon

   Insert 'points' or 'points_flip', according to the orientation of
   the polygon and generating the 2 subdomains polygon1 and polygon2

   return polygon1, polygon2
end
```

Below, in Listing 3.7, we present the implementation of the first algorithm, as described earlier. It generates two subdomains from a leaf quadrant and the corresponding control polygon of the NURBS curve, and searches for the kernel of the resulting subdomain.

Listing 3.7: Function *Algorithm1*

```
function [Quadtree] = Algorithm1(Quadtree,f)
   # Input: Quadtree (data structure), f (current leaf element)
   # Output: Quadtree (updated data tree with subdomains' kernels)

   # Calculate NURBS curve in current quad
   curve = calculateNURB;

   points = curve.CPs; # Store control points in variable
   points_flip = fliplr(points); # Flip the values of points

   # Find intersections of control polygon with quad
   Int = find_uniqueInt(Quadtree,f);

   # Store current quad vertices in variable
   Quad_current = Quadtree.QuadVertices

   # Initialize
   polygon = Quad_current;
   flag1 = 0;
   flag2 = 0;
   insertAfterPos = []; # Location of insertion of a vertex
   insertions = 0; # Tracking of number of insertions

   # Code to insert intersection points in quad polygon
   for l = 1:2 # Loop through the intersection points
      for i = 1:4 # Loop through the quad vertices
         Find where to insert the intersection points
         Insert intersection points in quad polygon
      end
   end

   # The locations to cut the quad polygon when inserting the vertices of
   # the control polygon
   cut1 = insertAfterPos(:,1)+1;
   cut2 = insertAfterPos(:,2)+1;

   # Inserting 'points' or 'points_flip' according to the orientation of
   # the polygon and generating the two subdomains polygon 1 and 2
   polygon1 = [polygon(:,(cut1):(cut2)), points(:,2:(length(points)-1))];
   polygon2 = [polygon(:,1:(cut1)), points_flip(:,2:(length(points)-1)),
            polygon(:, (cut2):end)];

   # Kernel search for polygon 1 and 2
   K1 = starShapedCheck(polygon1);
   K2 = starShapedCheck(polygon2);

   # Store K1 and K2 in Quadtree
   data = Quadtree.Node{f,1};
   data{12} = K1;
   data{13} = K2;
   Quadtree = Quadtree.set(f, data);
end
```

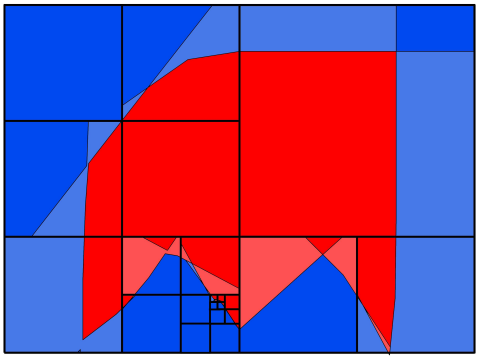
Algorithm 2 works with a polygonized approximation of the NURBS, and thus delivers more accurate results than *Algorithm 1* at the cost of increased computational time. This, however, is necessary when *Algorithm 1* is not applicable or fails to find a kernel. Another scenario when Algorithm 2 is needed, might be for the case that the control points, are not the seed points, used by the decomposition. In that case, it is also not allowed to use the control polygon for the kernel searching. By taking a better approximation of the NURBS curve, a kernel might be found, even if *Algorithm 1* has returned a negative result. In general, when comparing the kernel areas calculated by both algorithms, *Algorithm 2* delivers greater results. This however is not essential, as the goal is to find a location for a scaling center and thus *Algorithm 2* should only be used when needed. When *Algorithm 2* is required, the curve must be polygonised, before running the algorithm. The choice of number of line segments for the representation of the curve is a compromise between best approximation and fast runtime. Listing 3.8, below, shows the implementation of *Algorithm 2*.

Listing 3.8: Function *Algorithm 2*

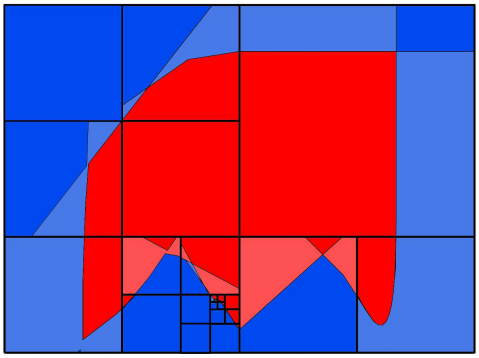
```
function [Quadtree] = Algorithm2(Quadtree,f)
   # Input: Quadtree (data strucutre), f (current leaf element)
   # Output: Quadtree (updated data tree with subdomains' kernels)

   # Calculate NURBS curve in current quad
   curve = CalculateNURBS;

   points = curve.CPs; # Store control points in variable
   points_flip = fliplr(points); # Flip the values of points

   # Find intersections of NURBS curve with quad
   Int = find_uniqueInt(Quadtree,f);

   # Store current quad vertices in variable
   Quad_current = Quadtree.QuadVertices

   [polygon1, polygon2] = starSplitPolygons(Quad_current,points,Int);
   K1 = starShapedCheck(polygon1);
   K2 = starShapedCheck(polygon2);

   # Store K1 and K2 in Quadtree
   data = Quadtree.Node{f,1};
   data{12} = K1;
   data{13} = K2;

   return Quadtree = Quadtree.set(f, data);
end
```

If *Algorithm 2* also fails, meaning at least one of the subdomains is not star-shaped, the given quad must be decomposed until all the resulting subdomains are star-shaped. Only when this condition is satisfied, the next step of the implementation can be processed. The following three figures represent a situation, where both algorithms would fail, and a decomposition would be needed. After the decomposition, Algorithm 1 finds kernels for all elements, and thus, like expected, Algorithm 2 would not be called. Figure 3.9 illustrates the results of the kernel-

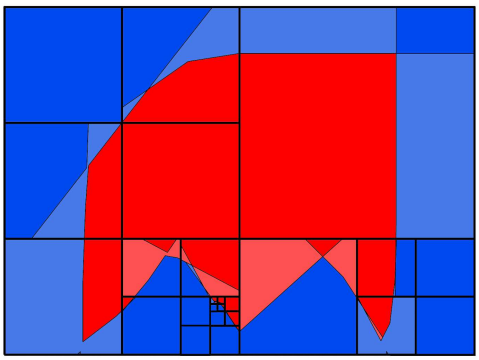
searching algorithms. The darker red and blue colors show the kernels inside and outside the interface, respectively, between the two subdomains. The translucent colors represent the rest of the subdomains, which are not kernels.



(a)



(b)



(c)

Figure 3.9: (a) Quadtree decomposition of control polygon, (b) Polygons 1 and 2 divided by the control polygon, (c) Kernels of the polygons

Figure 3.9(a) shows the situation after *Algorithm 1* is applied. One of the control points of the polygon at the bottom right part of the figure has its location outside of the quad, and thus no kernel is found for one of the subdomains in translucent blue. Figure 3.9(b) pictures the situation after applying *Algorithm 2*. The bottom-right quad is now divided by a polygonized NURBS curve, and, as shown, this has changed the situation in that quad, but the subdomain in translucent blue still has no kernel. A decomposition of this quad is therefore needed. Finally, Figure 3.9(c) visualizes the four new quads after the decomposition. For all the leaf elements, a kernel is found and the star-shape check is finished successfully, without applying *Algorithm 2*.

3.6 Quadtree-Balancing Procedure

Finding the neighbors of a given quad is necessary, in order to succesfully implement quadtree-balancing, as described in Section 2.2. This is not a trivial task, which offers several possible approaches. The one described by [4] is based on geometrical properties of the quadtree and can be applied for finding each neighbor. It is based on finding a common ancestor, tracing the geometrical path to its center and then retrace the path, making mirror image above an axis formed by the common boundary between the two quads. The first path is traced going upwards on the tree data structure, and the second going downwards. As here pointers to the parents and children of each quad are stored, this would be a valid approach.

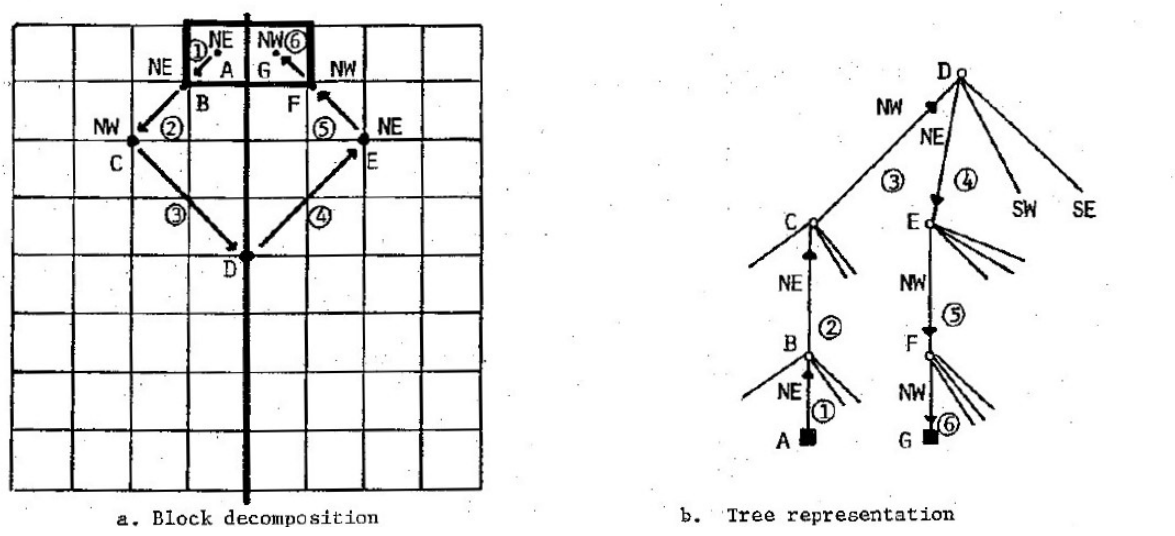


Figure 3.10: Finding an eastern neighbor of a quad [4]

Here, a more efficient approach is undertaken, and some known facts about the quadtree are taken advantage of. One can, for instance, guarantee the existence of at least two neighbors, since they must be two of the children of the same parent of the given quad. For instance, for a NW quad, the south and east neighbor must exist. For finding the other two neighbors a routine is created. This routine obtains the direction of the two eventual neighbors, taking advantage of some properties of the composition of the directions, which follows a pattern. Afterwards

the direction is searched on the address book (an array, which contains the directions of all quads) and if it exists, then is taken as the neighbor. Following that, it compares the level of subdivisions of the given quad and compares it with the one of its neighbors. If they differ in more than one, the decomposition function is called, and then the subsequent functions are also executed. This process of neighbor finding can be explained with the following example and illustration.

Consider the following to be coordinates of the NW quad: [22 22 11]. The coordinates of its east neighbor are [22 22 12], while the south neighbor is located at [22 22 21]. The coordinates of a possible west neighbor are [22 21 12], and coordinates of a possible north neighbor are [22 12 21]. One can notice, that the first requirement is to identify the common ancestor, which is [22 22] for the case of the east and south neighbors, and [22] for the eventual west and north ones. The rest of the remaining direction is obtained through the above-mentioned routine. In this case the function would not call the decompose function, as the 2:1 ratio requirement is fulfilled for the given quad.

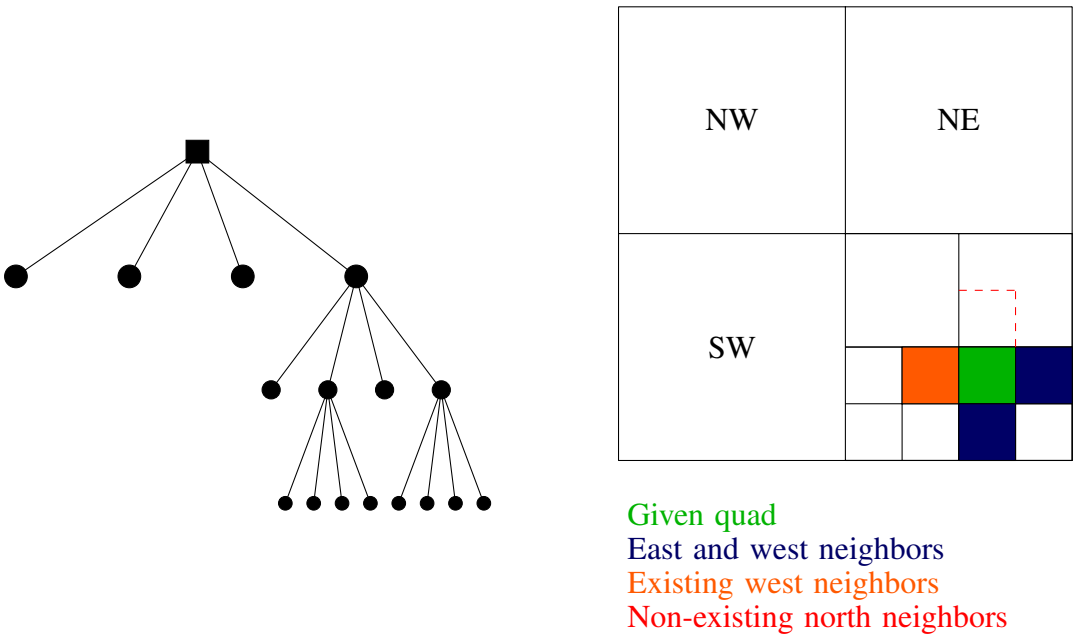


Figure 3.11: Quadtree decomposition of a square domain and its corresponding tree structure

In Listing 3.9, **function** *Balance_Quadtree* is presented, which obtains a balanced quadtree, based on unbalanced one. It runs through the nodes of the tree, checking how many subdivisons it has, and comparing them with the neighbors. If any quad is unbalanced, it calls the decomposition function. This is done recursively until the tree is balanced.

Listing 3.9: Function *Balance_Quadtree*

```
function [Quadtree] = Balance_Quadtree(Quadtree,NURBS, controlPoints,
                  knots, weights, degree, Boundary)
 # INPUT:
 # Quadtree data
 # Definition of the NURBS
 # OUTPUT:
 # Balanced Quadtree

 # Initialization
 # --------------

 # The function Location_Quads creates an adressbook of all nodes in 1:4
 # Format: (11 = 1, 21 = 2, 12 = 3, 22 = 4)
   Location = Location_Quads(Quadtree);


   while i <= length(Quadtree.Node)
      # loop over the nodes of the tree. The last two elements of its
      # direction determine if the quad is NW,SW,NE,SE
      Loc_Current = Location{i}; # Current quad's location in 1:4 format

      # Obtaining ammount of sons' generations (level of decomposition) of
      # current quad. This will be compared with it's neighbours. The
      # function number_sons takes the current location and the adressbook
      # 'Location' and gives as output the number of sons' generations
      [nSons] = number_sons(Location,Loc_Current);

      # If NW Quad. Check_ functions compares the number of son
      # generations for a current NW quad with its neighbours and eventually
      # calls the decomposition function
      if P == [1 1]
         [Quadtree, update_Location] = Check_NW(Quadtree, NURBS, controlPoints,...
         knots, weights, degree,i,Location,nSons,Loc_Current,Boundary);
      end
      # If SW Quad
      if P == [2 1]
         [Quadtree, update_Location] = Check_SW(Quadtree, NURBS, controlPoints,...
         knots, weights, degree,i,Location,nSons,Loc_Current,Boundary);
      end
      # If NE Quad
      if P == [1 2]
         [Quadtree, update_Location] = Check_NE(Quadtree, NURBS, controlPoints,...
         knots, weights, degree,i,Location,nSons,Loc_Current,Boundary);
      end
      # If SE Quad
      if P == [2 2]
         [Quadtree, update_Location] = Check_SE(Quadtree, NURBS, controlPoints,...
         knots, weights, degree,i,Location,nSons,Loc_Current,Boundary);
      end
   end
   i = i + 1;
end
```

Listings 3.10 and 3.11 show **function** *Check_NW* in two parts. This **function** looks for the

possible neighbors of a NW quad and compares their levels of decomposition. If they differ in more than one level, the 2:1 rule is not fulfilled, therefore it decomposes the quad further.

Listing 3.10: Function *Check_NW*

```
function [Quadtree,update_Location]=Check_NW(Quadtree, NURBS, controlPoints,...
knots, weights, degree,i,Location,nSons,Loc_Current,Boundary)
 # INPUT:
 # Quadtree data
 # Definition of the NURBS
 # Location: adressbook of all nodes
 # nSons: ammount of sons' generations of given quad
 # Loc_Current: adress of given quad
 # OUTPUT:
 # Quadtree after eventual decomposition
 # update_Location: flag for updating adressbook

 # Initialization
 # --------------
 # Pointer to father of actual quad
   Father = Quadtree.Node{Quadtree.Parent(i),1};

 # Adress of east sibiling
   Loc_E_Sib = Location{idx_E_Sib};

 # Checking its level of decomposition
   [nSons_E_Sib] = number_sons(Location,Loc_E_Sib);

 # If levels of decompisition differ in more than one, decompose
   if nSons_E_Sib > nSons + 1
      # Decompose_balance functions prepares input for decompose function
      [Quadtree] = decompose(Quadtree, NURBS, controlPoints, knots,
      weights, degree, auxliar_vars);
      break
   end
 # Adress of south sibiling
   Loc_S_Sib = Location{idx_S_Sib};
 # Checking its level of decomposition
   [nSons_S_Sib] = number_sons(Location,Loc_S_Sib);

 # If levels of decomposition differ in more than one, decompose
   if nSons_S_Sib > nSons + 1
      # Decompose_balance functions prepares input for decompose function
      [Quadtree] = decompose(Quadtree, NURBS, controlPoints, knots,
      weights, degree, auxliar_vars);
      break
   end
 # Address of eventual west sibiling, findig if it exists and level
 # of decompoition
   Loc_W_Sib = Location{idx_S_Sib};
 # Finding if sibiling exists
   for j = 1:length(Location)
      # Loops over adressbook, if exists flag=1
      tf = isequal(Location{j}, Loc_W_Sib);
      if tf == true; flag = 1; end
   end
```

Listing 3.11: Function *Check_NW* – Continued

```
  if flag == 1
     [nSons_W_Sib] = number_sons(Location,Loc_W_Sib);
     # If levels of decomposition differ in more than one, decompose
     if nSons_W_Sib > nSons + 1
        # Decompose_balance functions prepares input for decompose function
        [Quadtree] = decompose(Quadtree, NURBS, controlPoints, knots, weights,
        degree, auxliar_vars);
        break
     end
  end
# Address of eventual north sibiling, finding if it exists and level
# of decompoition
  Loc_N_Sib = Location{idx_N_Sib};

# Finding if sibiling exists
  for j = 1:length(Location)
     # Loops over adressbook, if exists flag=1
     tf = isequal(Location{j},Loc_N_Sib);
     if tf == true; flag = 1; end
  end

  if flag == 1
     [nSons_N_Sib] = number_sons(Location,Loc_N_Sib);
     # If level of decompisition differ in more than one, decompose
     if nSons_N_Sib > nSons + 1
        # Decompose_balance functions prepares input for decompose function
        [Quadtree] = decompose(Quadtree, NURBS, controlPoints, knots, weights,
        degree, auxliar_vars);
        update_Location = 1;
     end
  end
end
```

An example of quad balancing is given, by using the NURBS curve, shown in Figure 3.2. Figure 3.12 displays the quadtree decomposition output right before performing **function** *BalanceQuadtree*. Figure 3.13 displays the final (balanced) quadtree decomposition.

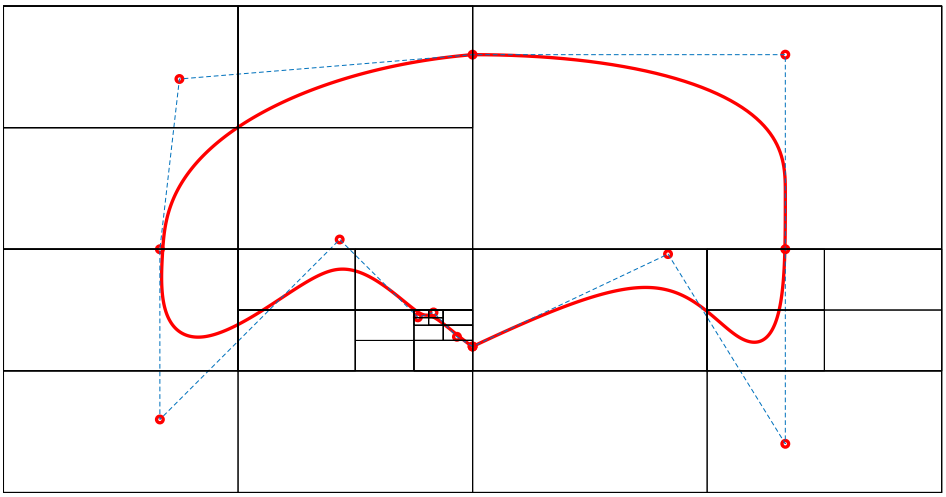


Figure 3.12: Grid decomposition before balancing

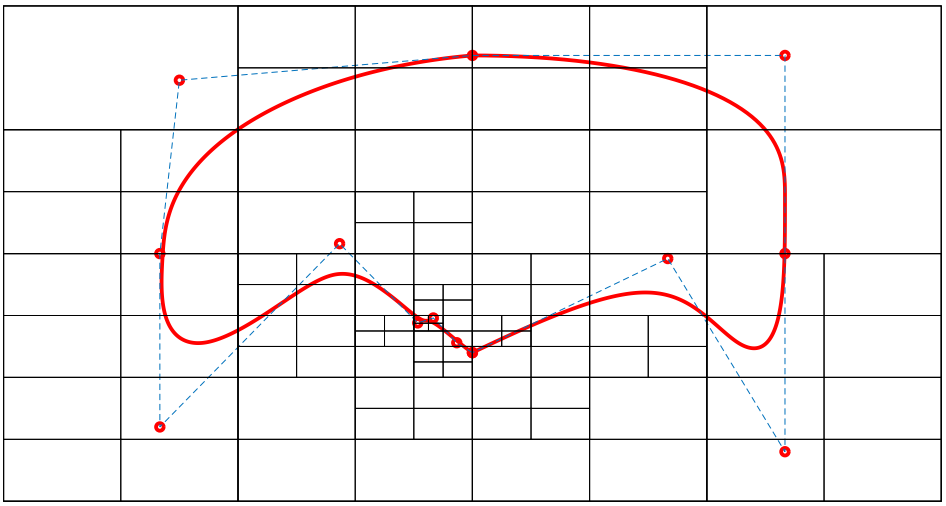


Figure 3.13: Grid decomposition after balancing

4 Examples

As mentioned in the introduction, in this section, a quadtree decomposition for different NURBS descriptions of an interface will be performed. The aim of this section is both to reproduce some expected shapes of material interfaces and to test the limits of the developed program.

4.1 Inclusion and Fibers

In this section, three different examples are discussed. The first two represent a geometry, which could be a void or an inclusion. The first one could represent an interface between a steel bar and concrete in a reinforced concrete section, while the third one could represent a fiber in a matrix.

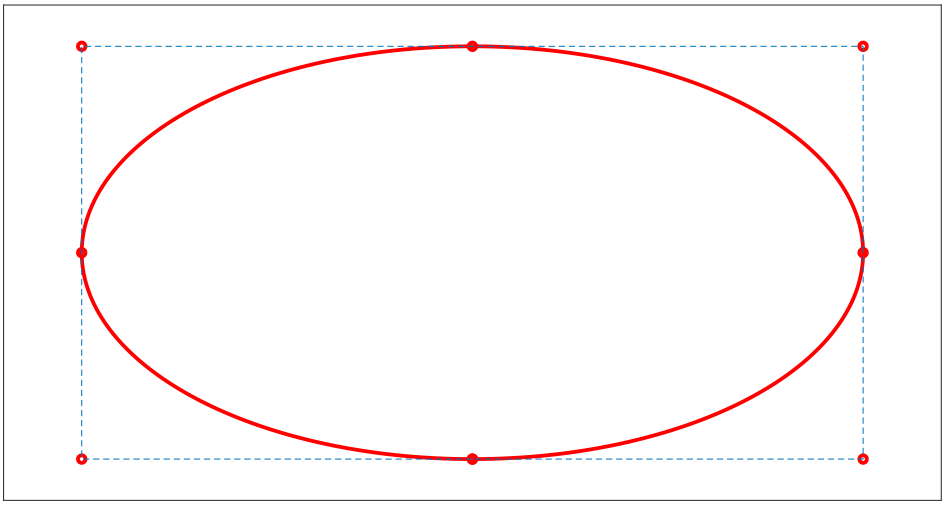


Figure 4.1: NURBS representation of an inclusion or void

Figure 4.1 displays a NURBS representation of a circular interface. After performing the quadtree decomposition, Figure 4.2 is obtained. Yet, this figure shows the state of the decomposition before balance or star-shape criteria are considered. However, as all resulting elements are star-shaped and the quadtree is balanced, no further decompositions are executed.

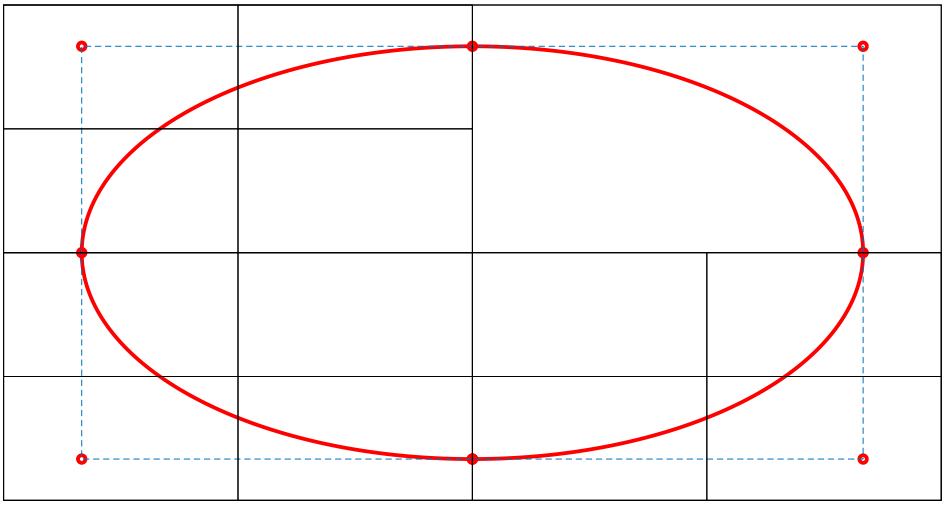


Figure 4.2: NURBS representation of an inclusion or void after quadtree decomposition

The next example to be shown is simillar to the first one. Here, the control points, controlling one of the main sections of the ellipsoid are pulled together, resulting in a more complex shape of the inclusion. Figure 4.3 illustrates this geometry.

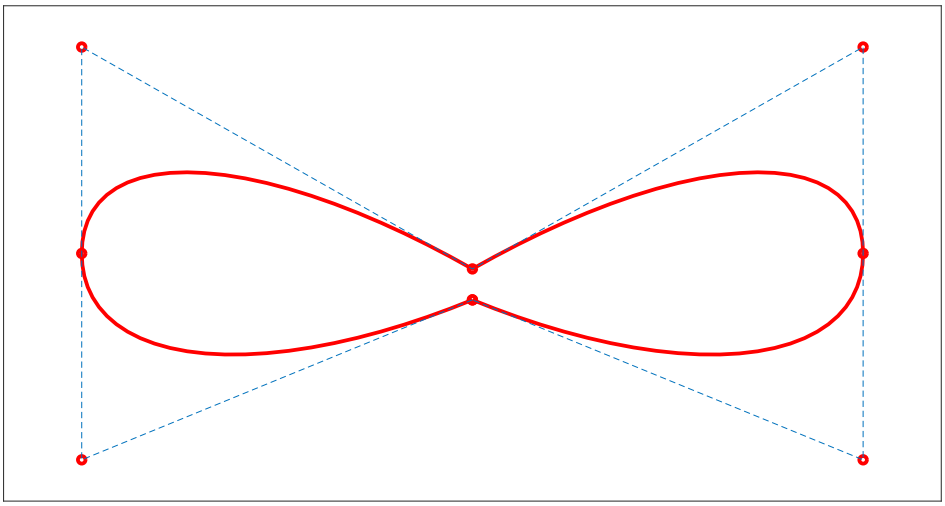


Figure 4.3: NURBS representation of the second shape of inclusion

Figure 4.4 pictures the unbalanced quadtree decomposition of the second example, but after star-shape check is perfomed. Figure 4.5 shows the quadtree decomposition after balancing is performed.

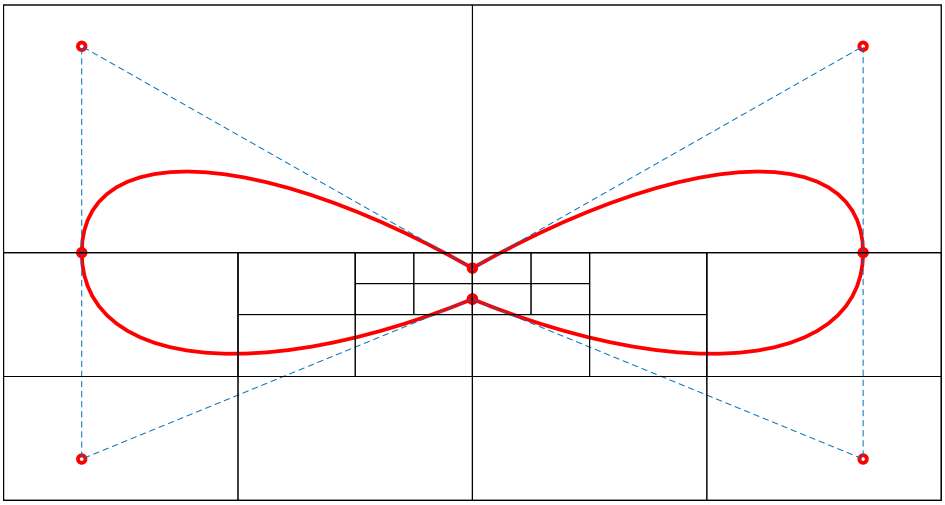


Figure 4.4: NURBS representation of the second shape of inclusion before balancing

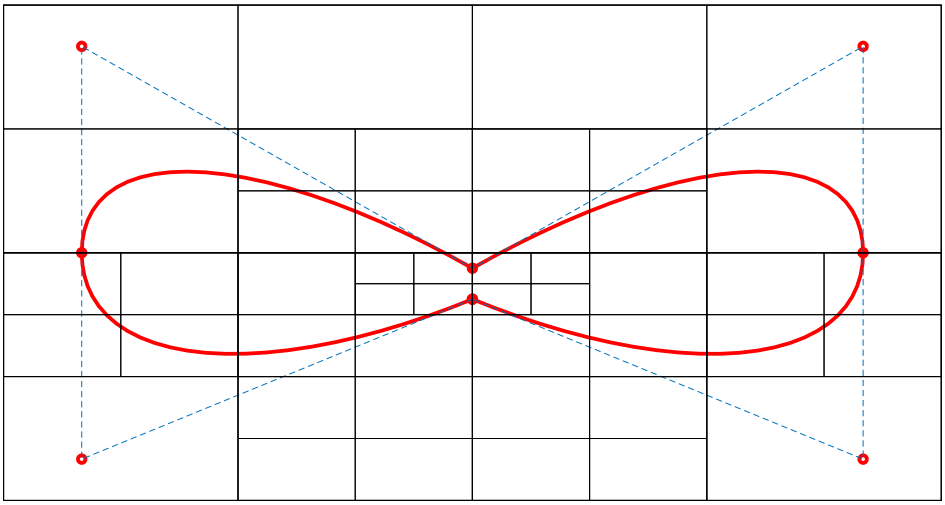


Figure 4.5: NURBS representation of the second shape of inclusion after balancing

The third is the flattened shape, which could represent a fiber inclusion. It is shown in Figure 4.6. The state of the quadtree decomposition after a star-shape check has been performed, and before balancing, is shown in Figure 4.7. Quadtree decomposition after balancing is shown in Figure 4.8.

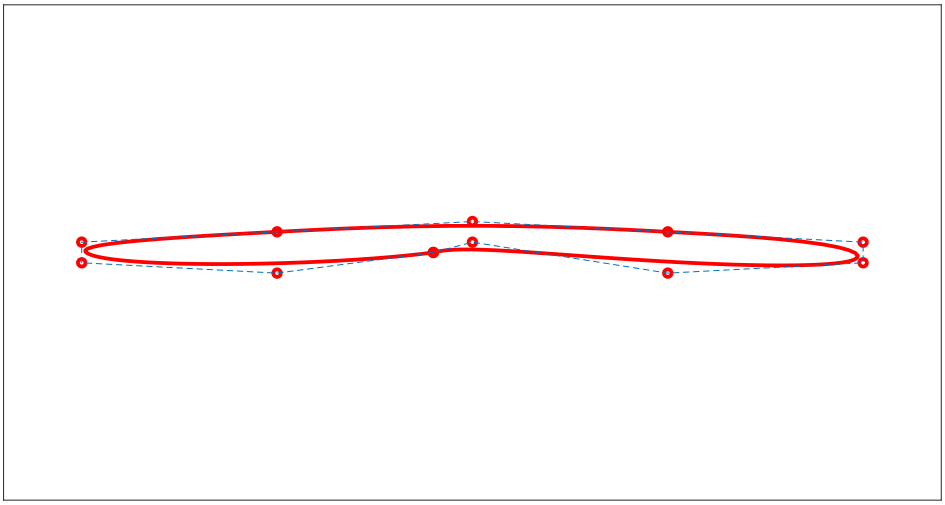


Figure 4.6: NURBS representation of a fibre in a matrix

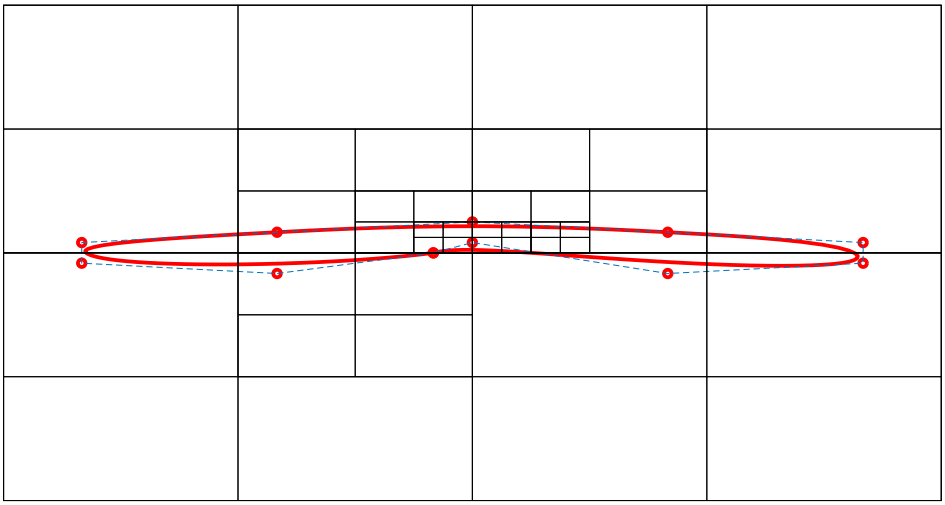


Figure 4.7: NURBS representation of a fibre in a matrix before balancing

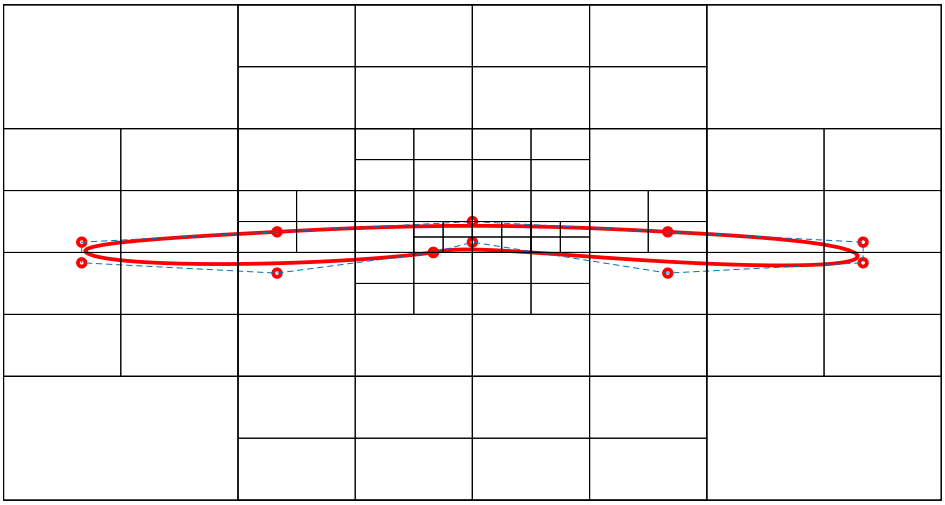


Figure 4.8: NURBS representation of a fibre in a matrix after balancing

4.2 Degenerate Case

In this subsection, two more examples are depicted. The first example, which is almost a non-Jordan curve, represents a limit case of the one despicted in Figure 4.3, while the second one is an actual degenerate case. The first example, displayed in Figure 4.9, appears tp be a self-intersecting curve. The middle control points are actually separated by a value of 10^{-5} at the input. This can be seen in Figure 4.10, along with a detail of the accuracy of the obtained NURBS-quad intersection, marked in blue, in Fig 4.11.

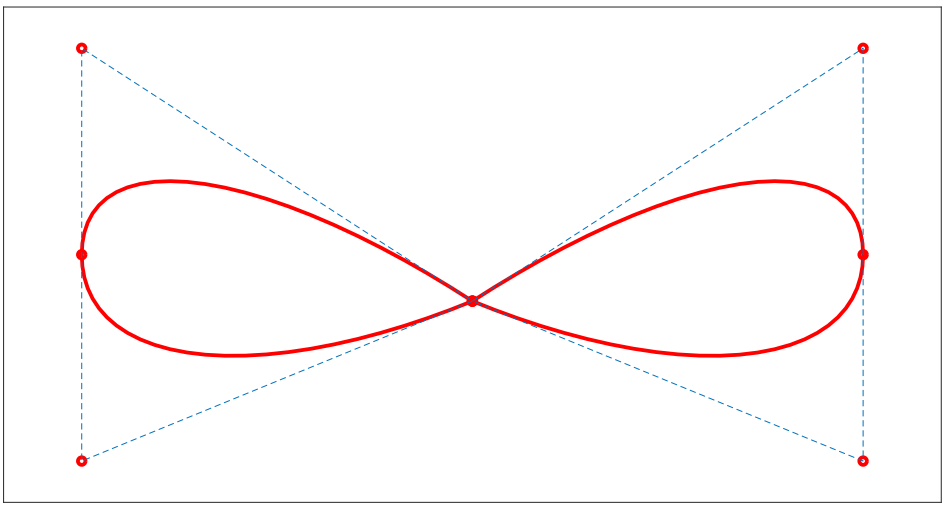


Figure 4.9: NURBS representation of the first degenerate example

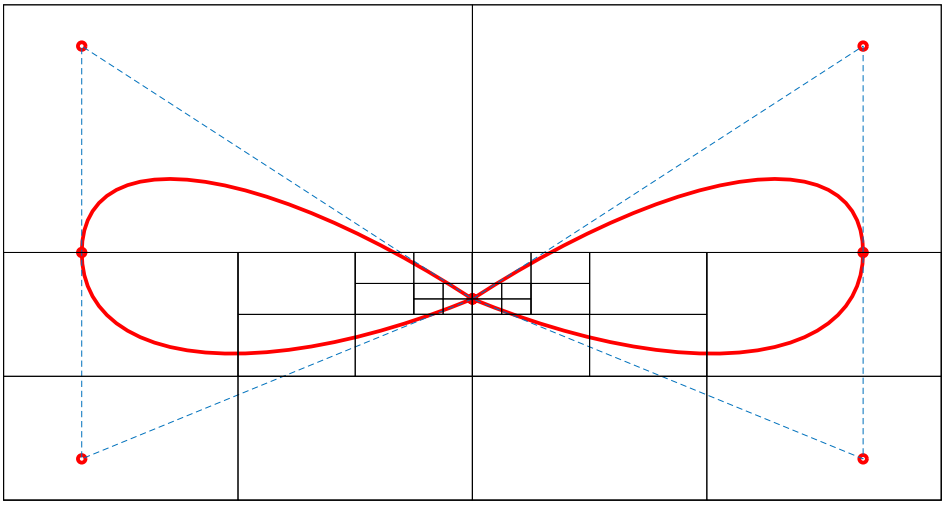


Figure 4.10: NURBS representation of the first degenerate case before balancing

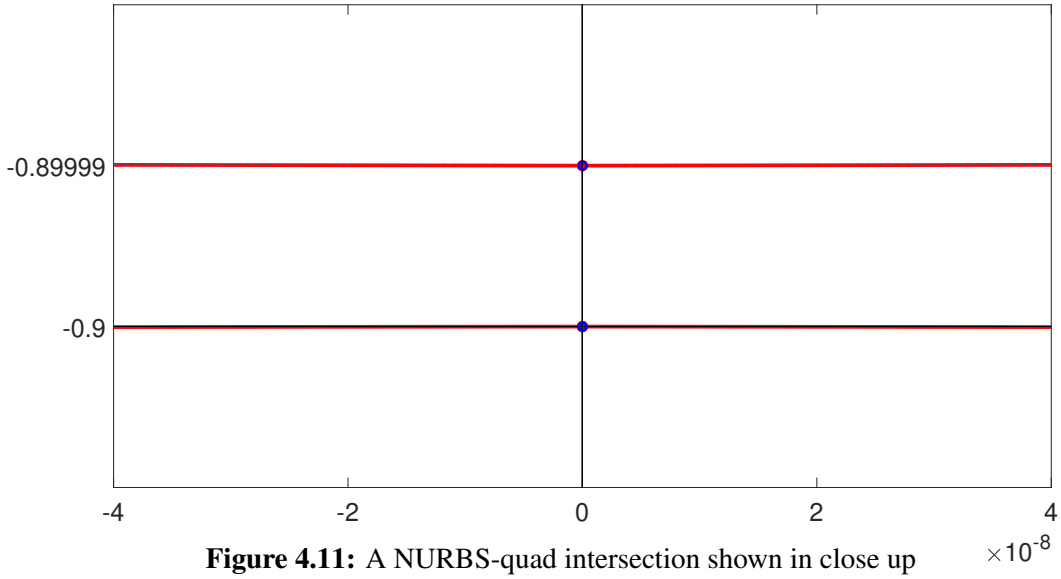


Figure 4.11: A NURBS-quad intersection shown in close up

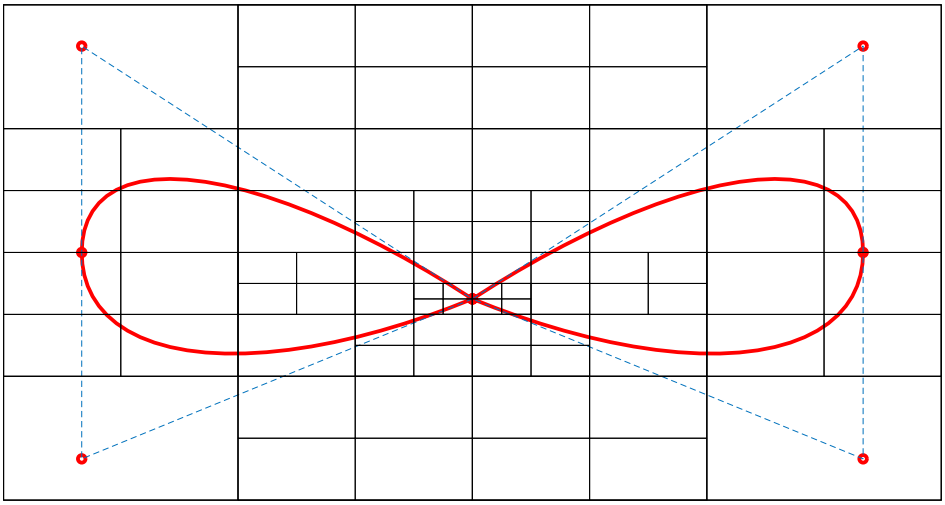


Figure 4.12: NURBS representation of the first degenerate case after balancing

The presented degenerate case represents a non-Jordan curve, since it is self intersecting. The aim here is to test the limit of the developed program. Since this case was not considered during the conceiving process, the computation of the quadtree decomposition is expected to fail. Figure 4.13 displays the NURBS definition of this particular interface, which looks similar to the last example.

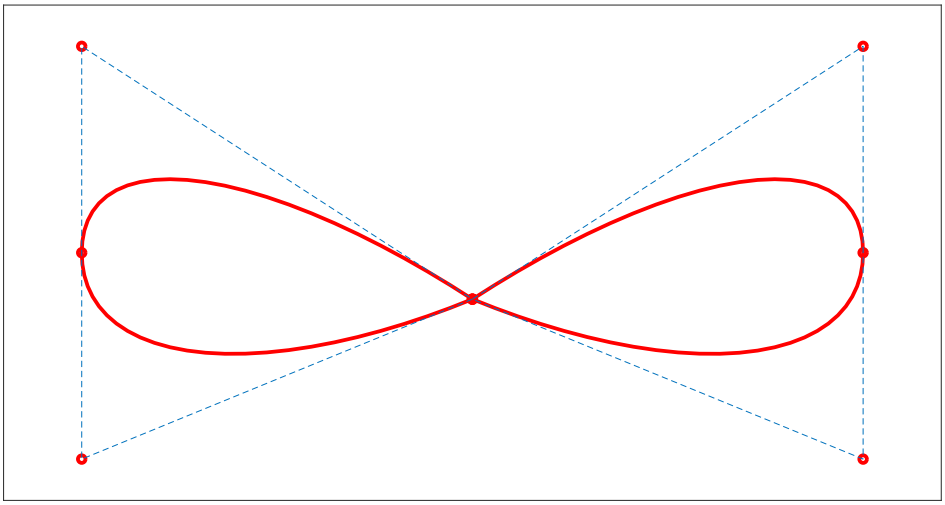


Figure 4.13: NURBS representation of the second degenerate case after balancing

Once the **function** *decompose* is called, it enters in an infinite loop. This is due to the maximum two intersections per quad condition, which cannot be achieved. This is because one has an unexpected amount of control points at the same physical location, and the algorithm cannot separate them in two different quadrants. This is depicted in Fig 4.14, where the NURBS–quad intersection points have been marked in blue, to show that **function** *decompose* does enter in an infinite loop at the location of the multiple control points.

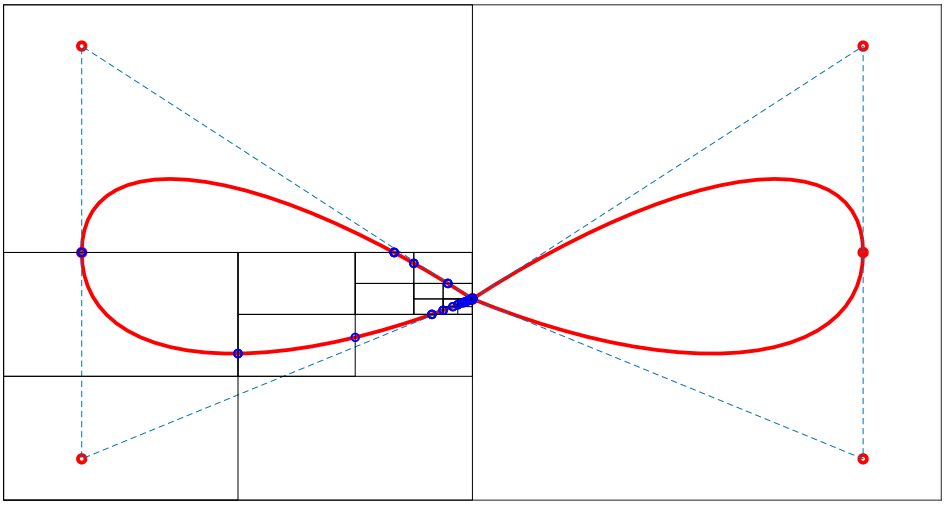


Figure 4.14: Intersection points between quads and the NURBS

There are several possible solutions to this problem. One could not use the control points as seed points, but then the decomposition process would not depend on the position of the control points. Another possibility would be to simply assign some of the multiple control points to another quad, and the decomposition process would continue. An additional solution would employ two tangent NURBS, representing two different interfaces. This would require the implementation of the quadtree decomposition, taking into account multiple interfaces, which is the natural next step for this project.

5 Conclusion

A quadtree decomposition of two subdomains in NURBS boundary representation is presented. The main tasks achieved are the development of a NURBS-calculating algorithm and performing a quadtree decomposition of a given domain, in which the resulting elements must be star-shaped. The development of the NURBS algorithm is mainly based on literature. NURBS are employed to represent the interface between two different type of materials. The quadtree decomposition algorithm serves several purposes at the same time. Its main function is to generate a grid over a given domain, while creating a tree data structure, that corresponds to the grid, and to store information in each node of the structure. If a section of the NURBS curve is enveloped in a leaf quad of the resulting decomposition, it will contain a section of the interface, and thus, two different type of materials can be identified. This leads to dividing the leaf element into two subdomains, which then, following the idea of the Scaled-Boundary Finite Element Method, must be checked for the shape of the polygons, which should be star-shaped. At the last section of the document, the validity of the obtained algorithms has been verified. Descriptions of different interfaces have been tested.

This scientific work can be used for further implementation of the SBFEM for composite materials using NURBS representation of the interfaces. Natural next developments of the project would be to consider self-intersecting curves, determining a method to deal with them, as well as considering a domain which contains several material interfaces. The use of MATLAB has benefited with regards to the readability of the code. The developed algorithms can concurrently also be easily translated into more suitable language for the needs of FE calculations, like Fortran, etc.

References

[1] J. A. Cottrell, JR T. Hughes & Y. Bazilevs: *Isogeometric analysis: toward integration of CAD and FEA*. Wiley and Sons, 2009

[2] L. Piegl & W. Tiller: *The NURBS Book*. Springer-Verlag Berlin Heidelberg, 1997

[3] M. de Berg, O. Cheong, M. van Kreveld & M. Overmars: *Computational Geometry*. Springer-Verlag Berlin Heidelberg, 2008

[4] H. Samet: Neighbor Finding in Quadtrees. In: *Proc.PRIP-81*, 1981, S. 68–74

[5] S. Klinkel & R. Reichel: A finite element formulation in boundary representation for the analysis of nonlinear problems in solid mechanics. In: *Computational Methods in Applied Mechanics and Engineering* 347 (2019), S. 295–315

[6] J. Zhao & X. Wang: An Algorithm to Searching for the Kernel of a Simple Polygon. In: *International Conference on Computer Application and System Modeling (ICCASM 2010)* Bd. 3, 2010, S. 455–457

[7] W. T. Sederberg & R. S. Parry: Comparison of three curve intersection algorithms. In: *Computer-aided design* 18.1 (1986), S. 58–63

[8] R. Kharian & L. A. Piegl: A Knowledge-Guided Approach to Line NURBS Curve Intersection. In: *Computer-Aided Design and Applications* 11.1 (2014), S. 1–9

[9] C. L Bajaj & X. Guoliang: NURBS approximation of surface/surface intersection curves. In: *Advances in Computational Mechanics* 2.1 (1994), S. 1–21

[10] Jean-Yves Tinevez: *Tree data structure as a MATLAB class*. – URL `http://www.mathworks.com/matlabcentral/fileexchange/35623-tree-data-structure-as-a-matlab-class`. – Zugriffsdatum: 10.04.2018

Selbstständigkeitserklärung

Hiermit erklären wir, dass wir diese Arbeit eigenständig und ohne unzulässige Hilfsmittel verfasst haben. Wir haben keine anderen als die angegebenen Quellen verwendet und alle Zitate kenntlich gemacht. Wir haben diese Arbeit vorher noch keiner anderen Prüfungskommission vorgelegt.

Aachen, den March 31, 2019

Benjamin Cerar
Mario Aguilar Rueda
Hristo Kovachev