## University of Padua

## DEPARTMENT OF MATHEMATICS "TULLIO LEVI-CIVITA" <br> Master degree in Mathematics

## Automatic algorithm for foreign object damages detection on engine compressor's blades

Candidate:<br>Thesis advisor:<br>Marco Agugiaro<br>1132552<br>Prof. Fabio Marcuzzi<br>Research supervisors:<br>Dr. Giulia Antinori<br>Julian von Lautz

## Contents

1 Problem description ..... 7
1.1 Data structure ..... 9
1.2 Damages classification ..... 11
2 State of the art ..... 13
2.1 Previous work on the problem ..... 13
2.1.1 Motivation and objective of the partitioning ..... 14
2.2 Splines functions and approximation ..... 15
2.2.1 Alternative approximating functions: polynomials ..... 16
2.3 Least square approximation ..... 17
2.3.1 Penalized splines ..... 17
2.3.2 Penalized least square approximation ..... 18
2.4 Strategy to detect the location of damages ..... 20
2.5 Damage classification ..... 20
2.6 Principal component analysis ..... 21
2.6.1 Sample Principal Component Analysis ..... 23
2.7 PCA applications: shape analysis and orientation ..... 24
3 Proposed solution ..... 27
3.1 Strategy for surface partitioning ..... 28
3.1.1 Airfoil partitioning ..... 30
3.1.2 Edges partitioning ..... 31
3.1.3 Top part ..... 35
3.1.4 Corners partitioning ..... 35
3.1.5 Fillet partitioning ..... 35
3.2 Approximating function choice ..... 36
3.3 Strategy for identifying potential location of the damages ..... 38
3.4 Damage classification ..... 38
3.5 Components analyzed ..... 39
3.6 Result structure ..... 40
3.6.1 Approximant parameters choice ..... 41
3.6.2 Thresholds choice ..... 42
3.6.3 Displayed results ..... 42
4 Application of the detection algorithm to blisk analysis ..... 45
4.1 Airfoil ..... 46
4.2 Edges ..... 49
4.2.1 First partitioning method ..... 51
4.2.2 Second partitioning method ..... 58
4.3 Summary ..... 64
5 Application of the algorithm to the analysis of worn blades ..... 65
5.1 Airfoil ..... 67
5.2 Edges ..... 70
5.2.1 Border ..... 71
5.2.2 Edge ..... 74
5.3 Top part ..... 74
5.4 Summary ..... 77
6 Conclusions and improvement ideas ..... 81
6.1 Summary ..... 81
6.1.1 Blisk summary ..... 81
6.1.2 Worn blades summary ..... 82
6.2 Outlook ..... 83
6.2.1 Partitioning method ..... 83
6.2.2 Approximating function construction ..... 84
6.3 Conclusions ..... 86
A Code ..... 89
A. 1 Main ..... 89
A.1.1 blisk_main_spline ..... 89
A.1.2 blade_main_spline ..... 89
A.1.3 blisk_main_poly ..... 90
A.1.4 blade_main_poly ..... 90
A. 2 Analysis ..... 90
A.2.1 analysis_spline_main ..... 90
A.2.2 analysis_spline ..... 94
A.2.3 analysis_poly_main ..... 101
A.2.4 analysis_poly ..... 105
A. 3 Partition ..... 109
A.3.1 partition_main ..... 110
A.3.2 partition_refinement ..... 115
A.3.3 partition_grid_cut ..... 121
A.3.4 utilities ..... 123

## Introduction

In this thesis will be presented an algorithm to analyze the surface of jet engine blades. The purpose of the algorithm is to locate damages and imperfections on the blade and to classify them. In particular, impact caused by foreign object damages, or FODs for brevity, will be distinguished by other forms of damages, like corrosion.

The informations collected can be used, for example, to develop new components more resistant and durable.

At the moment the detection task is manually performed, which is an extremely time consuming process. The result obtained are also subjective because the similar shape of FODs and other damages often makes it difficult to tell them apart. The algorithm proposed greatly reduce the time necessary for the inspection and the results returned lack any kind of subjectivity.

The first chapter contains the description of the problem, the characteristics of the data available and the feature of the damages to be found. The second chapter presents the mathematical tools used to develop the algorithm and previous work on the topic of damage detection.

The third chapter describes thoroughly the proposed method. The algorithm is based on the comparison between a scan of the surface and a smoothed version of it, through which anomalies are detected. Principal component analysis is then used to classify them.

Chapters four and five show the results obtained applying the algorithm to the blades in different states, newly manufactured and already used.

In chapter four newly produced blisk blades are analyzed to detect blemishes caused by the manufacturing process. Damages on those blades are fewer in number and all imperfection must be identified as relevant. In chapter five corrosion and deformations are also present due to the usage, making the detection harder and requiring the selection of only the relevant damages.

The conclusions are presented in chapter six, along with ideas on possible future direction of research. In the appendix a Python implementation of the method is available.

The method proposed in this work does not include neural networks, which are often used for image recognition and analysis purposes, and was developed according to the nature of the problem and the available data. The damages to be found, in fact, present themselves in a variety of shapes and sizes, and the blades themselves have different features depending on their model. The blades visually analyzed, while numerous, were not enough to guarantee a sufficiently big training set. In addition the subjectivity of the visual detection could cause problem with the training on the smaller damages.

The method proposed resulted effective, highly adaptable to different blade models, and it only requires the setting of a limited number of parameters to be properly applied.

As the results show, the method is not absolutely perfect, but it is considered satisfactory for the company. In particular it is worth highlighting that the algorithm is being actually applied by the company as an extra safety check for blisk blades. The use as an auxiliary tool for visual damage detection on worn down blades was also discussed.

## Chapter 1

## Problem description

Scope of this thesis is to present an algorithm to detect and classify damages on inner components of jet engines. The method presented aim to locate automatically the position of the impacts and analyze their shape.

Foreign object damages, shortened FODs, refers to damages on aircrafts, helicopters, launch vehicles engines or other aviation equipment which take place when a foreign object strikes the engines, flight controls, airframe and other operating systems [10]. For a more in-depth dissertation on the foreign object nature refer for example to [11] and for their specific effects to [22].

FODs are a serious issue in aviation maintenance industry that can only be properly controlled by performing regular and accurate controls over aircraft and engine components.

Engines in particular tends to draw small objects (and unlucky birds), ice and dust particles along with the large amount of air ingested to function. Any solid materials sucked in can impact with velocities in the range $100-350 \mathrm{~m} / \mathrm{s}$ depending on the rotational speed of the blades.

In this thesis we will deal specifically with engine blades and the dents left on them by FO, with focusing in particular on dents small enough to be hard to detect with direct visual inspections. Collected data on the shape, size and position of those indents is useful to better understand the FO dynamic in the engine and help design more resistant components.

To explain the interest in small damages must be remembered that deformations big enough to be noticed during regular controls are immediately substituted. Their prevention is mainly dependent on the the correct following of the safety procedures while the aircraft is on the ground [11]. The effects of smaller indents are less evident but have effect on longer term.

Turbine engines blades experience low-cycle fatigue (LCF) loading due to normal start-flightlanding operations and high-cycle fatigue (HCG) loading due to vibration and resonant airflow dynamics. The small surface indentation caused by FOs impacts can increase the speed of the wearing and become fatigue crack initiation sites [23].

To minimize those kind of failures the components must be substituted on a regular basis as a precaution, even when no damages are immediately apparent. This increases substantially the maintenance costs, prompting for the companies in the airline industry to research over this subject.

A better understanding of those damages position and effect would provide insights on how to improve the designs of blades and components in general, justifying the interest on the topic. Obtaining an extensive collection of data on FOD features and locations is a fundamental prerequisite for any study on their prevention.

Right now the detection and classification of these micro-impact on dismissed blades is performed by an operator checking on a 3D visual representation of the surface. The model is obtained from the physical blades through the use of white light scanner technology.


Figure 1.1: On the left (1.1a) a photo of one engine compressor blade, on the upper right (1.1b) a picture of an engine compressor. On the lower right (1.1b) a detail of damages on one corner.


Figure 1.2: Basic structure of a jet engine from [25]. Blades of different shapes are present in the fan, low and high pressure compressor, low and high pressure turbines.

This visual analysis is an extremely time-consuming process and various days of work can be spent to analyze a single blade. Those time-frames are necessary to perform an accurate search on the magnified image of the blades. The reasons are the small dimensions of the damages, in the tenth of millimeters, and the need to look at the model from different angles to detect all the imperfections on it.

An automatic algorithm would not only save time but also give less subjective results.
Before defining the characteristics of the damages, it can be useful to understand the structure of the available data and their simpler properties.

### 1.1 Data structure

The last years saw a sharp increase in the use of laser scanners to measure a physical object and build a 3D model of it. There are a variety of applications for such technology, for instance to detect imperfections (as in our case) or to digitalize and mold the structure of a new component to build. Depending of the application there are different ways to both measure the object and translate them into a 3D structure. The scans involved in this work were obtained using a white light scan measuring system to produce a triangular mesh as a stl file.

Stl files describe unstructured triangular meshes, where each triangle composing it is described using its vertexes and the unit normal to its surface. If the normal is not given it is automatically evaluated to be orthogonal to the triangle surface. Let us suppose that the points are distributed on a horizontal plane, the orientation of the normal depends on the order of the vertexes. Using the right hand rule the normal will point upwards if the vertices are given in a counter-clockwise order, downwards otherwise.

The structure of the mesh is richer than the cloud of points formed by the triangles vertices. The triangles and their normal give a oriented piecewise linear surface and also a proximity


Figure 1.3: White light scanner mounted on top of a robotic arm and one blade to be measured. Picture from [24].
relation between the vertexes.
As an example the normal direction allow the distinction between inside and outside of a surface, to understand if an imperfection is protruding or indented. The triangles structure can be used to build a graph connecting vertexes of the same triangles. This graph can then be used to easily select neighborhood of points without measuring the distances.

Working with the scan of one blade, no matter how accurate the scanner can be, implies to deal with an approximated representation of the object and not a perfect one. Measurement includes a certain level of noise, and can be imprecise in on certain parts of the object, for the blades on the thinner parts of edges where the mesh loses its smoothness. The data involved in this work presented a noise level of 0.001 mm , one order of magnitude smaller than the minimum depth of a relevant damage ( 0.02 mm ), giving a tight safety margin.

To maintain a error level small enough to detect the damages the order of magnitude of the number of points involved is in the millions. This implies high costs in terms of memory and computational time for every analysis involving the whole mesh.

About the distribution of mesh vertices is interesting to notice that the point distribution along the surface is not uniform in the data used. The mesh construction starts from distinct measurement taken from different angles and return a mesh with a higher density of points in the areas with greater curvature and less points on the flatter portions. This allow for a smoother result by keeping the angle between adjacent faces normal under a threshold without using all the measurement data.

The coordinate system is set to be the same for every blade, using a bolt near the base as a reference point. This rototranslation of the whole mesh allow the data to be standardized, at least in a macroscopic sense.

This procedure is performed separately and won't be discussed in this work, but is important to notice that translating and rotating the mesh is not sufficient to fit it to a reference model.

Precisely because the blades analyzed are worn it can not be expected for them to be identical. Trying to use a pristine one as a reference model shows differences in the blade shape much bigger than the damage size. The causes of those deformations are the mechanical stress the blades are subject and the impacts close to edges and corners.

The combined effect of those forces causes small bends to appear on the thinnest parts of the blades without predictable patterns.

In addition small pieces on edges, corners and topmost portion get occasionally chipped off


Figure 1.4: Worn blade mesh scan, corner detail. FODs are highlighted in the green, in the red examples of corrosion and yellow marks some dubious cases.
because of erosion and grazing impacts. This means that is not possible to find reliable reference points usable to identify those deformation. Performing non-rigid transformations to adjust those small deformation without changing the shape of the damages is then difficult.

The aligned meshes then have to be assumed similar but with small differences, mainly on the thinnest parts. The scans are too different from each other to use a common reference model of the blade to make a comparison.

### 1.2 Damages classification

Detecting FOD on blades does not mean to locate every imperfection on the surface. Even excluding the noise involved in the measuring process, impacts are not the only source of damages.

Another source of defects to be kept into consideration is corrosion, which gradually erodes the surface giving it an orange peel appearance. The cavities caused by corrosion are extremely similar in size to the impact ones and differs mainly on their shape. ${ }^{1}$

The analysis should be able to distinguish between the two causes of damages. Visually the difference between impact and corrosion is usually clear, but there are no known criteria or parameter for this specific task.

Qualitatively speaking impact damages are rounder, deeper and more defined while corrosion is wider, more shallow and often elongated and branched with irregular patterns. Those unfortunately are only guidelines, sometimes corrosion run deeper than small FOD, or the impact is inside a corroded part.

On the edges of the blade the impacts are instead more elongated and scratch-like, given the different type of impact involved [21].

The shape of the potential FOD is the main discriminant, simply looking at their dimensions, length width and depth, does not usually allow for a safe distinction. What we look for are then the "rounder" and deeper impacts, while everything else will be classified as corrosion.

One common feature for both impacts and corrosion is that the depth is usually one order of magnitude smaller than length and width of the damage. This detail is important during the

[^0]analysis of the damages shapes.
Once again is important to highlight the small difference between the two kind of damages in both dimensions and shape. The minimum size of a relevant impact is extremely small, with a depth in the tens of micron, similar to the corrosion.

The accuracy requirements explain why non-rigid transformations of the blades are risky, and why small deformations of less than a millimeter spread along the length of the whole blade present such a big obstacle.

## Chapter 2

## State of the art

The problem of identifying and measuring damages or imperfections on scans of physical objects is surely not new. Various works which propose possible solutions has been published, in both general and specific cases. However, the requirement for this specific application are too restrictive to apply one of the available methods.

The algorithm proposed here is similar to the one in [2] mainly regarding the methods used to classify the structure of potential damaged zones. The identification of the damaged zones is instead an original idea.

In this chapter the algorithm presented in the article is described to give an idea on how this type of problem is solved in a different field of application. The mathematical methods used both there and in our solution will also be shown, while the algorithm we propose will be discussed in the next chapter.

### 2.1 Previous work on the problem

The article [2] describe an algorithm to detect damages on wind turbines. The authors propose a method based on the comparison between scans of the objects and a 3D reference model of a pristine turbine to locate the damages. The imperfections identified are then distinguished between impacts, cracks and corrosion.

The most remarkable difference between the application discussed in the article and the one of this thesis is the size of the analyzed objects.

While jet engine blades are even less than 10 centimeters big the size of a wind turbine varies from 40 to 90 meters in diameter [7] with blades up to 80 meters long [8]. This difference has consequences over the level of deformation acceptable and the size of the regular damages. A bending of 0.1 millimeters is enough to prevent the use of a reference model for engine blades, the same level of deformation is instead irrelevant for wind turbines.

For the same reason the size of damages of interest is vastly different, but once re-scaled they should present roughly the same shape and features.

Despite the differences between applications, the fundamental ideas on how to classify the imperfections presented in [2] are also useful for our problem, with due modifications.

Getting back to the algorithm, the authors first divide the surface in smaller patches and build a parametric approximation of each of them. Those approximating surfaces are then compared to the corresponding portion of the reference model. The model is also defined piecewise on the same portions of the object, and with the same kind of approximating function.

By evaluating the differences between the two surfaces the locations of the damages are identified as the locations where the differences are higher than a minimum threshold. The


Figure 2.1: In green the points to be modeled, the grid represent the approximant. The result in 2.1 c , with only half the surface, is clearly more accurate to the one with superimposition of points (2.1a) or almost vertical sides (2.1b).
focus is then shifted to classifying the imperfections based on their type, specifically impacts, cracks and corrosion. This is done by considering both the dimensions of the clusters of points composing the damages and the principal component analysis of the differences on those same points.

The necessity of those intermediary steps is discussed in the following sections, starting from the reason behind the initial partitioning.

### 2.1.1 Motivation and objective of the partitioning

While it is possible to build a local approximation of even complex shapes, if the graph of a function is used to parametrize it (as in both the article and our case) the surface structure must be simple enough. Horizontal folds or (almost) vertical parts of the meshes make the mesh impossible to be modeled with this kind of approximant as shown in Fig:2.1.

Working with smaller windows is useful to simplify the geometry of the portion involved and to decrease computational cost of many operation.

For the same reason each patch should be oriented to reduce the presence of steep inclines as much as possible.

An example could be the difficulty of parameterizing a folded surface compared to the same task performed on the separated layers. This is exactly what happens on the edges of the blade's faces. Such portions should be divided and a local approximation build, after applying proper rotations.

The problem then is to decide how the partitioning should be performed to simplify the structure as much as possible and to minimize the problems associated with the parametrization.

The procedure have to also be automatized, so this choice have to be independent of the small variations that appear in the scans and in this regard adaptable once set for a class of objects. This issue is not discussed further in the article, the solution proposed can be found in the next chapter.

Once the partitioning has been performed, and the patches are oriented in a proper way, the authors use spline functions to build a parametric approximation of them.

### 2.2 Splines functions and approximation

Given an interval $[a, b] \subset \mathbb{R}$ close and limited and given $a=x_{1} \leq x_{2} \leq \cdots \leq x_{n}=b$ a spline of degree $m$ is a $C^{m-1}([a, b])$ that, in each $\left[x_{i}, x_{i+1}\right] \forall i \in\{0,1, \ldots, n-1\}$ is a $m$ degree polynomial. The $x_{i}$ points are called knots.

A B-spline function, or basis spline, is a spline with minimal support with respect to his degree and the domain partitioning. To allow a B-spline to be defined also on the edges of a interval, the knots are usually padded by repeating the extremes $a$ and $b$ a number of time equal to the degree desired.

With an interval defined as above, B-splines are recursively defined as

$$
\begin{array}{cc}
B_{i, 0}(t)=\left\{\begin{array}{ll}
1 & x_{i} \leqslant t \leqslant x_{i+1} \\
0 & \text { otherwise }
\end{array}\right\}, & 1 \leqslant i \leqslant n  \tag{2.1}\\
B_{i, j}(t)=\frac{t-x_{i}}{x_{i+j}-x_{i}} B_{i, j-1}+\frac{x_{i+j+i-t}}{x_{i+j+1}-x_{i+1}} B_{i+1, j-1}, & 1 \leqslant i \leqslant n+m-j .
\end{array}
$$

where $B_{i, j}(t)$ is the B -spline centered in $x_{i}$ of degree $j$ and $t$ is a point of the domain.
We define spline functions of $d$ degree as linear combinations of $d$ degree B-splines, and can be seen as

$$
\begin{equation*}
s(t)=\sum_{i=1}^{n} B_{i, d}(t) c_{i} \quad \forall t \in[a, b] \tag{2.2}
\end{equation*}
$$

where $c_{i}$ are called control points, whose scalar values are associated with each of the B-splines. They can be seen as the amount of "pulling" or "pushing" performed on each knot, with $s$ being the resulting curve.

Since splines are polynomials piecewise, in virtue of the following theorem any continuous function can be uniformly approximated by them.

Theorem 2.1 (Weierstrass approximation theorem). Suppose $f$ is a continous real-valued function defined on the real interval $[a, b]$. For every $\epsilon>0$, a polynomial $p$ exist such that, for all $t$ in $[a, b]$, we have $|f(t)-p(t)|<\epsilon$.

To improve the accuracy of a polynomial approximant is necessary to increase its degrees. With splines the degree can be kept fixed changing instead the number of control points (and knots). This is particularly useful for interpolation purposes, where increasing the degree of polynomials, even piecewise, can produce oscillating results as shown with the Runge function example.

Going from two to three dimension, B-splines are defined as the tensor product between two-dimension ones. This means that, given a domain $[a, b] x[c, d] \subset \mathbb{R}^{2}$, a 3D B-spline on the $(i, j)$ knot of the grid produced by crossing a knot vector $\left\{x_{0}, \ldots, x_{n_{x}}\right\} \subset[a, b]$ over another knot vector $\left\{y_{0}, \ldots, y_{n_{y}}\right\} \subset[c, d]$, of degree $d_{x}$ along the first axis and $d_{y}$ along the second, is defined as

$$
B_{\left(i, d_{x}\right),\left(j, d_{y}\right)}^{2}(u, v)=B_{i, d_{x}}(u) B_{j, d_{y}}(v)
$$

Their linear combination is then in the form

$$
\begin{equation*}
s_{c}(u, v)=\sum_{i=1}^{n_{x}} \sum_{j=1}^{n_{y}} B_{i, d_{x}}(u) B_{j, d_{y}}(v) c_{i, j} \quad \forall t \in[a, b] \tag{2.3}
\end{equation*}
$$

where the control points are still associated to the various B-splines.
The parameters necessary to define a function like $s_{c}(u, v)$ are then the number and position of the knots, the degrees of the splines along each axis and the values of the control points.


Figure 2.2: From left to right: 2D B-spline basis for 2nd degree splines, 3D B-spline basis with bi-variate 2nd degree splines and example of spline surface.

Increasing the number of control points, and of the respective B-splines, increase the degree of freedom of the spline function, allowing a potential better fit when the spline is used as an approximating function.

Once the knots are placed and the degrees are set, is possible to find the values of the control points to approximate a set of points by minimizing the energy functional

$$
\begin{equation*}
E(c)=\sum_{k=1}^{N}\left(\sum_{i=1}^{n_{x}} \sum_{j=1}^{n_{y}} B_{i, d_{x}}\left(u_{k}\right) B_{j, d_{y}}\left(v_{k}\right) c_{i j}-z_{k}\right)^{2} \tag{2.4}
\end{equation*}
$$

with $P_{k}=\left(u_{k}, v_{k}, z_{k}\right)$ measured points on the surface. Minimizing the square values of the difference is a natural way to obtain a good approximation, and while not the only method possible, is certainly the most widespread.

Is also possible to build the B -spline basis on knots that are not equispaced, to increase the accuracy in certain part of the domain. In our implementation the position of the knots was determined in the same way as for the value of the control points.

Splines are not the only possible choice as an approximating functions, even polynomials can be used for the same purpose.

### 2.2.1 Alternative approximating functions: polynomials

A simpler alternative to spline approximants are polynomials. Our datasets are composed of surfaces in a 3D space so is natural to work with bivariate polynomial. From now on by polynomials of $m$ by $n$ degrees, or $m \mathrm{x} n$ for short, we mean

$$
\begin{equation*}
p(x, y)=\sum_{i=0}^{m} \sum_{j=0}^{n} c_{i j} x^{i} y^{j} \quad(x, y) \in \mathbb{R}^{2}, c_{i j} \in \mathbb{R} \tag{2.5}
\end{equation*}
$$

where the constants $c_{i j}$ unequivocally define it. To approximate a set of $N$ points with a polynomial one possible solution is minimizing the energy functional

$$
\begin{equation*}
E(c)=\sum_{k=0}^{N}\left(\sum_{i=0}^{m} \sum_{j=0}^{n} c_{i j} x_{k}^{i} y_{k}^{j}-z_{k}\right)^{2} \tag{2.6}
\end{equation*}
$$

over $c=\left(c_{00}, \ldots, c_{m n}\right)$.
In both (2.6) and (2.4) the most natural way to compute the values of the parameters is the least square method.

### 2.3 Least square approximation

Suppose $f: \mathbb{R}^{d} \longrightarrow \mathbb{R}$ is an unknown function and that we are given a set of measurements $\left(x_{i}, z_{i}\right)=\left(\left(x_{1}, \ldots, x_{d}\right)_{i}, z_{i}\right)$ for $i=1, \ldots, N$, where $z_{i}=f\left(x_{i}\right)+\epsilon_{i}$ with $\epsilon_{i}$ measurement error. We want to build an approximating function $s \in S$ where $S$ is the subspace spanned by the functions $B_{1}, \ldots, B_{k}$ (with $k \ll N$ ). We can then express $s$ in function of a parameter vector $c=\left(c_{1}, \ldots, c_{k}\right) \in \mathbb{R}^{k}$ as

$$
\begin{equation*}
s=s_{c}=\sum_{j=1}^{k} B_{j} c_{j} . \tag{2.7}
\end{equation*}
$$

To define $c$ with the linear least square method we choose the values that minimize the energy functional

$$
\begin{equation*}
E(c)=\frac{1}{2} \sum_{i=1}^{N}\left(s_{c}\left(x_{i}\right)-z_{i}\right)^{2}=\frac{1}{2} \sum_{i=1}^{N}\left(\sum_{j=1}^{k} B_{j}\left(x_{i}\right)-z_{i}\right)^{2} \tag{2.8}
\end{equation*}
$$

that is equivalent to find the solution for

$$
\begin{equation*}
\underset{c \in \mathbb{R}^{d}}{\operatorname{argmin}}\|B c-b\|_{2} \tag{2.9}
\end{equation*}
$$

where $B_{i j}=B_{j}\left(x_{i}\right)$ and $b_{i}=z_{i}$.
Sometimes can be useful to add some regularizing components to avoid excessive oscillation in the resulting approximating function, or to just get a smoother solution. One way to do this is by using a penalized version of linear least square approximation.

### 2.3.1 Penalized splines

To obtain a smoother spline approximant is possible to add a weight term to (2.4) to control a specific feature of the resulting function. Ss a penalty term we could for example take the thin plate spline energy as in [6]:

$$
\begin{equation*}
J(c)=\int_{a_{1}}^{b_{1}} \int_{a_{2}}^{b_{2}} s_{x x}^{2}(u, v)+2 s_{x y}(u, v)+s_{y y}(u, v)^{2} d v d u \tag{2.10}
\end{equation*}
$$

where $s=s_{c}$. This can be expressed, once the (2.3) is substituted, as

$$
\sum_{i=1}^{n_{x}} \sum_{j=1}^{n_{y}} \sum_{r=1}^{n_{x}} \sum_{s=1}^{n_{y}} E_{i j r s} c_{i j} c_{r s}
$$

where

$$
\begin{aligned}
& E_{i j r s}=A_{i j r s}+2 B_{i j r s}+C_{i j r s}, \\
& A_{i j r s}=\int_{a_{1}}^{b_{1}} B_{i, d_{x}}^{\prime \prime}(u) B_{r, d_{x}}^{\prime \prime}(u) d u \int_{a_{2}}^{b_{2}} B_{j, d_{y}}(v) B_{s, d_{y}}(v) d v, \\
& B_{i j r s}=\int_{a_{1}}^{b_{1}} B_{i, d_{x}}^{\prime}(u) B_{r, d_{x}}^{\prime}(u) d u \int_{a_{2}}^{b_{2}} B_{j, d_{y}}^{\prime}(v) B_{s, d_{y}}^{\prime}(v) d v, \\
& C_{i j r s}=\int_{a_{1}}^{b_{1}} B_{i, d_{x}}(u) B_{r, d_{x}}(u) d u \int_{a_{2}}^{b_{2}} B_{j, d_{y}}^{\prime \prime}(v) B_{s, d_{y}}^{\prime \prime}(v) d v .
\end{aligned}
$$

How such weight can be included during the research of the control points is discussed in the next subsection, along with the importance of the correct choice of $\lambda$.

### 2.3.2 Penalized least square approximation

Least square fitting is useful for its simplicity and for the noise filtering property [4], coming from the disparity between number of data and parameters. To get a smoother result, to further reduce the noise or to control particular features to $s$, a penalty term $J(c)$ can be added to (2.8) before minimizing the result.

Many smoothing terms can be expressed as $J(c)=c^{T} E c$ where $E$ is a symmetric non-negative matrix in $\mathbb{R}^{k \times k}$. This result is a new energy functional in the form

$$
\begin{equation*}
E_{\lambda}(c)=\frac{1}{2} \sum_{i=1}^{N}\left(\sum_{j=1}^{k} B_{j}\left(x_{i}\right)-z_{i}\right)^{2}+\frac{1}{2} \lambda c^{T} E c \tag{2.11}
\end{equation*}
$$

where $\lambda \geq 0$ is the weight applied to the penalty(or smoothing) term.
This allow the following:
Definition 2.2. The penalized least squares fit of the function $f$ based on data $\left(x_{i}, z_{i}\right)$ for $i=1, \ldots, N$ is the function $s_{c(\lambda)}$ where $c(\lambda)$ minimize $E_{\lambda}(c)$.

Under the minimal hypothesis that $B$ has full rank the following holds
Theorem 2.3. For any $\lambda \geq 0$ there exist a unique vector $c(\lambda)$ minimizing the functional $E_{\lambda}(c)$ in (2.11). In particular, $c(\lambda)$ is the unique solution of the system

$$
\begin{equation*}
\left(B^{T} B+\lambda E\right) c=B^{T} b \tag{2.12}
\end{equation*}
$$

Proof. Setting the gradient of $E_{\lambda}(c)$ equal to zero gives (2.12) and the condition on $B$ ensure that $G=B^{T} B$ is symmetric, positive definite and non singular. Given that $E$ is symmetric and non-negative by definition we get that $B^{T} B+\lambda E$ is also symmetric and non-negative defined, so the solution exist and is unique.

Let's now consider the mean square error of the fit

$$
T_{(x, z)}(\lambda)=\frac{1}{N} \sum_{i=1}^{N}\left[s_{c(\lambda)}\left(x_{i}\right)-z_{i}\right]^{2}
$$

and see how the value of $\lambda$ influence it. In case of data not affected by noise, with $\epsilon_{i}=0 \forall i$ holds
Theorem 2.4. The function $T_{(x, z)}(\lambda)$ is monotone increasing $\lambda \geq 0$ with $\dot{T}_{(x, z)}(0)=0$ and $\lim _{\lambda \longrightarrow \inf } \dot{T}_{(x, z)}(0)=0$.

This means that in absence of noise the smoothing of the approximant can only maintain or worsen the error level. However the data available are almost never noiseless. In the simplest case possible, where $\epsilon_{i}$ are normally distributed with mean 0 and variance $\sigma^{2}$, we have

$$
T_{\epsilon,(x, z)}(\lambda)=T_{(x, z)}(\lambda)+2 \epsilon^{T} A(\lambda)^{T}(A(\lambda) c-b)+\epsilon^{T} A(\lambda)^{T} A(\lambda) \epsilon
$$

where $A(\lambda)=B(G+n \lambda E)^{-1} B^{T}$, whose mean value is

$$
\begin{equation*}
\varepsilon T_{\epsilon,(x, z)}=T_{(x, z)}(\lambda)+\frac{\sigma^{2} A^{2}(\lambda)}{n} \tag{2.13}
\end{equation*}
$$

The behavior of this new energy function is described by the following


Figure 2.3: A surface example with one damage as the indent in the center and localized noise to simulate corrosion. The base structure is the one of $-\frac{x^{2}}{10}+\frac{y^{2}}{10}$ polynomial.


Figure 2.4: The upper row shows the spline approximations of Fig:2.3 without penalty (2.4a) and with $\lambda=10(2.4 \mathrm{c})$. The penalized case does not model the damage, the regular one partially fits the indent. In the lower row the difference between approximation and data for the two cases. The penalized case 2.4 d , being less flexible, has higher error on the border and slightly higher differences in the impact.

Theorem 2.5. The function $\varepsilon T_{\epsilon,(x, z)}$ has value $\varepsilon T_{\epsilon,(x, z)}(0)=T_{(x, z)}(0)+\frac{k \sigma^{2}}{n}$ and asymptotically approach the value $T_{(x, z)}(\mathrm{inf})+\frac{k \sigma^{2}}{n}$ as $\lambda \longrightarrow \mathrm{inf}$. It's derivative is negative for $\lambda=0$ and

$$
\begin{equation*}
\varepsilon T_{\epsilon,(x, z)}(\lambda)-\varepsilon T_{\epsilon,(x, z)}(0) \geq \frac{\sigma^{2}}{n}(t(\lambda)-t(0)) \tag{2.14}
\end{equation*}
$$

where $t(\lambda)=\operatorname{trace}\left(A^{2}\right)$.
This shows that in presence of noise $\lambda=0$ is not the optimum and a higher value can lead to better results. Unfortunately the only way to determine a good choice of $\lambda$ is by testing it on multiple values, requiring a trial-and-error strategy.

### 2.4 Strategy to detect the location of damages

In article [2] the location of the damages is found using the computed parametric approximation of the patches and with a spline reference model of the same portions analyzed. The two functions are compared by computing the absolute value of the differences between corresponding control points.

The differences are evaluated only on the control points, the original points of the mesh are not considered anymore. It's then clear the necessity of a number of control points sufficient to ensure that every damage is not only modeled, but also has enough of them to be properly analyzed. The exact amount of points depend on the size of relevant damages: the smaller the minimum acceptable damage, the more control points are required.

Working only with the control points fasten this part of the analysis but the computational price is paid searching the approximant. An high enough number of control points would make the combined cost of finding the approximant and performing the damage analysis higher than using the initial point directly.

The difference values are all non-zeros because of small error of measurement, alignment or approximation. To remove this noise the differences must be filtered with a minimum threshold level to remove this kind of noise.

The choice of this threshold is particularly important: too small and undamaged parts will be identified as damaged by the algorithm spoiling the quality of the following analysis, too high and small damages will be ignored or will have too little points to be properly analyzed.

The list of differences between control points is divided between the ones with a variation too small, that get rounded down to zero, and the ones with the relevant difference. The latter are the only one that will be considered in the following parts.

Once separated, by some proximity or connectivity criteria, the points forming the damages need to be studied separately for each distinct cluster to determine their type and importance.

### 2.5 Damage classification

To determine what type of damage is being analyzed various parameters can be checked.
The easiest and most natural features to be extracted are:

- damage area,
- maximum distance between two point of one cluster, as an indicator of how "long" one damage is,
- depth as the maximum value for the differences.


Figure 2.5: Example of difference between data and model, on the left unfiltered on the right cropped.

This is not enough to distinguish the causes of the damage, but already gives an idea on how big (and then dangerous) one imperfection is.

The next step is to perform an indirect form of measurement by using the principal component analysis. Using the filtered differences as inputs it is possible to extract an indication of the shape of the indent through the normalized vector of the principal component values.

To better understand how PCA can help detecting the shape of a cluster of points it's continuous and discrete formulations are presented.

### 2.6 Principal component analysis

Suppose that $\boldsymbol{x}$ is a (vertical) vector of $p$ random variables and the structure of variance and covariance is of interest. The covariance matrix is defined as

$$
\begin{equation*}
\Sigma=E\left[(\boldsymbol{x}-\boldsymbol{\mu})(\boldsymbol{x}-\boldsymbol{\mu})^{t}\right] \tag{2.15}
\end{equation*}
$$

where $\boldsymbol{\mu}$ is the mean value vector of the $p$ variables and $E$ is the expected value operator. The variance $\operatorname{Var}(\boldsymbol{x})$ of the variables is the diagonal of $\Sigma$ matrix, to obtain the variance of $x_{i}$ we can then compute $\operatorname{Var}\left(x_{i}\right)=\boldsymbol{e}_{i}^{t} \Sigma \boldsymbol{e}_{i}$ with $\boldsymbol{e}_{i} \in \mathbb{R}^{p}$ equal to zero on every entry except for the $i$-th where is equal to one. Using $\Sigma$ and the variance is possible to compute the correlation matrix, which component definition is

$$
\begin{equation*}
\operatorname{corr}_{i, j}(x)=\frac{\Sigma_{i, j}(x)}{\operatorname{Var}_{i}(x) * \operatorname{Var}_{j}(x)} \quad \forall 1 \leq i, j \leq p \tag{2.16}
\end{equation*}
$$

Unless in some simple cases, looking at the covariance matrix usually is not really helpful to understand the variables dynamic. Another method to obtain better information is necessary.

Principal component analysis (PCA) is a statistical method that returns linearly uncorrelated variables from potentially correlated ones by using orthogonal transformations. The new variables are called principal components $(\mathbf{P C})$, are orthogonal to each other and, in decreasing order for the highest variance along the direction orthogonal to the previous. A detailed description of the following part can be found in [5].


Figure 2.6: Examples of damages shown in [2], on the top-left impact damage, on the top-right crack damage and corrosion on the bottom, with respective normalized PCA values.

To do that we seek for a linear transformation $l_{1}(x)=\boldsymbol{\alpha}_{1}^{t} \boldsymbol{x}$, where $\boldsymbol{\alpha}_{1}=\left[\alpha_{1,1}, \ldots, \alpha_{1, p}\right] \in \mathbb{R}^{p}$, such that the variance $\operatorname{Var}\left(\boldsymbol{\alpha}_{1}^{t} \boldsymbol{x}\right)=\boldsymbol{\alpha}_{1}^{t} \Sigma \boldsymbol{\alpha}_{1}$ is maximized. Then for $\boldsymbol{\alpha}_{2}$ uncorrelated with the previous $\boldsymbol{\alpha}_{1}$ with the same property of maximum variance. We repeat the process choosing new $\boldsymbol{\alpha}_{i}$ uncorrelated from all the previous, until $\boldsymbol{\alpha}_{p}$ included. To avoid infinite results for the $\alpha_{i}$ values the condition $\boldsymbol{\alpha}_{i}^{t} \boldsymbol{\alpha}_{i}=1$ is imposed.

To maximize $\operatorname{Var}\left(\boldsymbol{\alpha}_{1}^{t} \boldsymbol{x}\right)$ subject to $\boldsymbol{\alpha}_{1}^{t} \boldsymbol{\alpha}_{1}=1$ we can use the Lagrange multipliers maximizing

$$
\begin{equation*}
\boldsymbol{\alpha}_{1}^{t} \Sigma \boldsymbol{\alpha}_{1}-\lambda\left(\boldsymbol{\alpha}_{1}^{t} \boldsymbol{\alpha}_{1}-1\right) \tag{2.17}
\end{equation*}
$$

where $\lambda$ is a Lagrange multiplier. Differentiating with respect to $\boldsymbol{\alpha}_{1}$ and looking for the zero in the derivative gives

$$
\left(\Sigma-\lambda \boldsymbol{I}_{p}\right) \boldsymbol{\alpha}_{1}=0
$$

where $\boldsymbol{I}_{p}$ is the $(p x p)$ identity matrix. This imply that $\lambda$ is an eigenvalue of $\Sigma$ corresponding to the eigenvector $\boldsymbol{\alpha}_{1}$. To decide which couple of eighnvector and eigenvalue to choose we must remember that we were maximizing

$$
\boldsymbol{\alpha}_{1}^{t} \Sigma \boldsymbol{\alpha}_{1}=\boldsymbol{\alpha}_{1}^{t} \lambda \boldsymbol{\alpha}_{1}=\lambda
$$

so we need the maximum eigenvalue of $\Sigma$, and the corresponding eigenvector is $\boldsymbol{\alpha}_{1}$. The second PC, which maximize $\operatorname{Var}\left(\boldsymbol{\alpha}_{2}^{t} \boldsymbol{x}\right)$ is subject to $\boldsymbol{\alpha}_{2}^{t} \boldsymbol{\alpha}_{2}=1$ and uncorrelated to $\boldsymbol{\alpha}_{1}$ that means that

$$
0=\boldsymbol{\alpha}_{2}^{t} \Sigma \boldsymbol{\alpha}_{1}=\boldsymbol{\alpha}_{2} \lambda \boldsymbol{\alpha}_{1}=\lambda \boldsymbol{\alpha}_{2} \boldsymbol{\alpha}_{1}=0
$$

thus one of the following must hold:

$$
\begin{equation*}
\boldsymbol{\alpha}_{1}^{t} \Sigma \boldsymbol{\alpha}_{2}=0 \quad \boldsymbol{\alpha}_{1}^{t} \boldsymbol{\alpha}_{2}=0 \tag{2.18}
\end{equation*}
$$

Choosing the second one and using the Lagrange multipliers we obtain

$$
\begin{equation*}
\boldsymbol{\alpha}_{2}^{t} \Sigma \boldsymbol{\alpha}_{2}-\lambda\left(\boldsymbol{\alpha}_{2}^{t} \boldsymbol{\alpha}_{2}\right)-\phi \boldsymbol{\alpha}_{1}^{t} \boldsymbol{\alpha}_{2} \tag{2.19}
\end{equation*}
$$



Figure 2.7: Examples of use of PCA to determine the axis of a point cloud representing an ellipsoid.
if we differentiate with respect to $\boldsymbol{\alpha}_{2}$ looking for the maximum, and multiply on the left for $\boldsymbol{\alpha}_{1}^{t}$ what we want to solve is

$$
\boldsymbol{\alpha}_{1}^{t} \Sigma \boldsymbol{\alpha}_{2}-\lambda\left(\boldsymbol{\alpha}_{1}^{t} \boldsymbol{\alpha}_{2}\right)-\phi\left(\boldsymbol{\alpha}_{1}^{t} \boldsymbol{\alpha}_{1}\right)=0
$$

but the first two terms are equal to zero because of (2.18) so $\phi$ must be zero. Then, removing the third term from (2.19), we obtain the same as (2.17) but for $\boldsymbol{\alpha}_{2}$ that, combined with (2.18), implies that $\boldsymbol{\alpha}_{2}$ is the eigenvector relative to the second bigger eigenvalue of $\Sigma$, and orthogonal to $\boldsymbol{\alpha}_{1}$. The procedure is similar for the following PC.

This can be done by computing the SVD decomposition of the covariance matrix: the PC values $\lambda_{i}$ here then the diagonal entries of the S matrix and the PC directions are the column of V.

However the data available are not continuous variables but a discrete set of points, requiring then a different definition of the variance matrix.

### 2.6.1 Sample Principal Component Analysis

Suppose that we have $n$ indipendent observation on the $p$-elements of the random vector $\boldsymbol{x}$, denoted as $\boldsymbol{x}^{1}, \ldots, \boldsymbol{x}^{n}$. Let's define $\boldsymbol{z}_{i, 1}=\boldsymbol{a}_{1}^{t} \boldsymbol{x}$ and choose the vector $\boldsymbol{a}_{1}$ to maximize the sample variance

$$
\begin{equation*}
\Sigma_{n}=\frac{1}{n-1} \sum_{i=1}^{n}\left(\boldsymbol{z}_{i, 1}-z_{1}\right)\left(\boldsymbol{z}_{i, 1}-z_{1}\right)^{t} \tag{2.20}
\end{equation*}
$$

where $z_{1}=\frac{1}{n} \sum_{i=1}^{n} \boldsymbol{x}_{i, 1}$ is the sample mean value and $\boldsymbol{x}_{i, 1}$ the first element of the $i$-th sample vector.

The result obtained in the previous section still hold for the sample formulation.
Principal component analysis can still be performed by finding eigenvectors and eigenvalues of the covariance (or correlation) matrix of the set of data. To do that the singular value decomposition can be used.

To prove that is useful to remember that the covariance matrix is symmetric positive semi-
definite. In fact

$$
\begin{align*}
y^{t} \Sigma_{n} y & =y\left(\frac{1}{n-1} \sum_{i=1}^{n}\left(\boldsymbol{z}_{i, 1}-z_{1}\right)\left(\boldsymbol{z}_{i, 1}-z_{1}\right)^{t}\right) y= \\
& =\frac{1}{n-1} \sum_{i=1}^{n} y^{t}\left(\boldsymbol{z}_{i, 1}-z_{1}\right)\left(\boldsymbol{z}_{i, 1}-z_{1}\right)^{t} y=  \tag{2.21}\\
& =\frac{1}{n-1} \sum_{i=1}^{n}\left(\left(\boldsymbol{z}_{i, 1}-z_{1}\right)^{t} y\right)^{t}\left(\left(\boldsymbol{z}_{i, 1}-z_{1}\right)^{t} y\right)= \\
& =\frac{1}{n-1} \sum_{i=1}^{n}\left(\left(\boldsymbol{z}_{i, 1}-z_{1}\right)^{t} y\right)^{2} \geq 0 \quad \forall y \in \mathbb{R}^{\prime}
\end{align*}
$$

and

$$
\begin{align*}
\Sigma_{n}^{t} & =\left(\frac{1}{n-1} \sum_{i=1}^{n}\left(\boldsymbol{z}_{i, 1}-z_{1}\right)\left(\boldsymbol{z}_{i, 1}-z_{1}\right)^{t}\right)^{t}  \tag{2.22}\\
& =\frac{1}{n-1} \sum_{i=1}^{n}\left(\left(\boldsymbol{z}_{i, 1}-z_{1}\right)\left(\boldsymbol{z}_{i, 1}-z_{1}\right)^{t}\right)^{t}=\Sigma_{n}
\end{align*}
$$

The spectral theorem then state that the matrix is diagonalizable using the eigenvectors as an orthonormal base i.e. $\Sigma_{n}=Q \Lambda Q^{t}$ with $Q Q^{t}=Q^{t} Q$ the identity matrix and $\Lambda$ diagonal matrix with the eigenvalues $\lambda_{1} \geq \lambda_{2} \geq \ldots \geq \lambda_{p} \geq 0$ as its diagonal entries $d_{11} \geq d_{22} \geq \ldots \geq d_{p p}$.

Computing the singular value decomposition will return $\Sigma_{n}=U \Lambda V^{t}=Q \Lambda Q^{t}$ then $U=V=$ $Q$ and $\Lambda=S$. The diagonal entries of $S$ are then the eigenvalues of $\Sigma_{n}$ and the rows of $V^{t}$ of the columns of $U$ are the corresponding eigenvectors. The PCA results can then be extracted from the singular value decomposition of the covariance matrix.

### 2.7 PCA applications: shape analysis and orientation

The principal components directions are an alternative reference system which has the property of having the greatest variance along the first axis, the second greatest (orthogonal to the first) on the second and so on. In this new coordinate system the variables are distinguished based on variance and this, for example, helps to view the data from a better angle.

To classify one finding as impact damage the first two components, representing length and width of the cluster of points, should be similar to select the ones round enough. The third component, alias the depth, should be smaller but still way bigger than what instead happens on the corroded locations, where the damage is lager and shallower.

The relative size of the damages dimensions guarantee that the depth is always the smaller of the three dimension for every imperfection, corrosion included ${ }^{1}$.

These qualitative consideration must be quantified and adapted depending on the problem at hand, to consider the shape and size of the damage to be found.

The third PC value also gives us an indication of the relative magnitude of the depth compared to length and width. As shown in [12] some indicators can be easily extracted from the PC values

Linearity: $\quad L_{\lambda}=\frac{\lambda_{1}-\lambda_{2}}{\lambda_{1}}$
Planarity: $\quad P_{\lambda}=\frac{\lambda_{2}-\lambda_{3}}{\lambda_{1}}$
Sfericity: $\quad S_{\lambda}=\frac{\lambda_{3}}{\lambda_{1}}$
Change of curvature : $\quad C_{\lambda}=\frac{\lambda_{3}}{\lambda_{1}+\lambda_{2}+\lambda_{3}}$
and the general shape of the object can be recovered from them.
This obviously does not mean that the results of PCA are meaningful on any set of data: outliers and more complex shapes can spoil the quality of the results as shown in Fig:2.8. Selecting only the point belonging to the damages will be necessary to perform a correct analysis.

[^1]

Figure 2.8: Example of use of PCA and its limitations, the set of data must be chosen correctly to get useful results. Damages must be delimited correctly to distinguish them between each other and to exclude undamaged portion of the surface.

## Chapter 3

## Proposed solution

The process described in the previous chapter, while suitable for wind turbines, could not be applied directly to jet engine blades because of two main issues.

First of all the damage considered in the article are way bigger than the one on jet engine blades. Even more so compared to the precision level attainable with the scanners because the margin of error is way smaller than what appear to be in the article. This does not only means that all the thresholds values must be kept extremely low, but also that the smaller variation in them can make a considerable number of damages appear (or disappear).

In addition the article method would require a parametric approximation capable of modeling the damages. The construction of such an approximating surface would be computationally expensive and difficult, requiring a number of control points close to the number of original points in the mesh. This fact would nullify the computational advantage of working with control points only, as building an adequate approximation would be too computationally expensive.

The second, and bigger problem is the lack of a usable reference model. Even if scans of newly produced blades are available, the small differences caused by consumption, manufacturing process and scanning conditions are bigger than the searched damages.

Furthermore the wide range of possible deformations makes the prospect of adapting the shape of the blade to a reference model impractical. Edges and corners, the natural reference points of the mostly flat blades, could be missing small pieces, or the scan itself could be inaccurate because of their thinness, making them unreliable in this sense. This makes the identification of such anomalies complicate and make their correction even harder.

So, while definitely not impossible to build a reference model and defining a procedure to adjust each blade to it, such a solution was discarded in favor of what appeared to be a more promising one.

To overcome this obstacle the solution found is to build an approximation of the surface, but an extremely simplified and smooth one. This smooth function will then be used as a local reference model and compared directly with the original points instead of the control points.

The small size of the damages is actually helpful in this, making them easily ignored in favor of the rest of the patch. Building an artificial comparison model for each window solves the problem of the bends, because the resulting artificial model is independent from the original structure.

The price to pay is the cost of actually finding a new approximant for each distinct portion of every distinct blade and working with all the points of the mesh.

In fact this process is actually fast considering that the requirement of a smooth approximation instead of an extremely accurate one require a low level of flexibility. An extremely low number of parameters are necessary compared to what would be needed to get an accurate approximation of the surface as in [2].

The use of small patches also keep the number of points used at one time extremely low, resulting in a reasonably fast process, as will be shown in the following chapters.

This is obviously far from being a solution to the problem. In addition to the parameters discussed earlier the typology of approximant and the way the portioning is done influence heavily the quality of the results.

The simpler and smoother the approximant, the more it can be used as a reference, but then the portion of surface involved must have a simple geometry too. This point must be considered in presence of fast variation of the curvature, steep vertical portion of the windows, and folds in general.

### 3.1 Strategy for surface partitioning

Knowing the shape dependency of the partitioning strategy the approach to the various part must be different depending on how such portion is structured. We can differentiate five main structures in the blades:

- airfoil: the large, almost flat central part;
- edges: the lateral borders of the blade, where the surface fold on itself;
- corners: on the top left and right, here the surface is extremely thin and potentially chipped;
- top: the top horizontal edge.
- fillet: the part joining base and blade, important for structural integrity, presenting a far greater curvature than the airfoil part.

Those parts must be divided from each other, and further separated in pieces small enough to be easy to work with.

To do this at first a rough partitioning is applied to the mesh dividing it into quadrilateral windows and then those parts are further refined. The dimension of such windows must be selected to retain enough points to allow an approximation capable of avoiding the damages, and small enough to avoid excessive change in shape and curvature.

The first subdivision is made through a grid, as in the figure above, starting from six points provided by the user. Four of them are chosen so that the blade over the fillet is fully included, the last two are the lower corners necessary to include the fillet. A lax bounding box would not cause significant problem to the algorithm provided that the first and last column of each row fully include the edges and corners.

Given the curvature of the blades, dividing the mesh in this simple way do not gives patches of the same size or even with the same height-width ratio. The procedure is not sensitive to small variation in this sense. The focal point is to separate different features (i.e. airfoil and edges) and keep the windows dimension close to a certain size. A greater uniformity in the patches form could help but is far from necessary.

Once those separations are done the boxes containing different parts of the blade can be distinguished based on their position in the grid. The corners are in the topmost left and right boxes, the edges in the first and last column, the top edge is in the topmost row, the fillet in the two lowest rows and the airfoil fill all the others.

The sub-meshes extracted are then further refined depending on the features contained.

(a)

(b)

(c)

(d)

Figure 3.1: On the left the grid used for the first subdivision step, on the right three different kind of results: edge (3.1b), airfoil (3.1c) and corner (3.1d) examples. The blue points in 3.1a are the one necessary to build it, the black dots represent a subset of matrix points. The grid contains the mesh but does not require the inclusion to be extremely tight. The two lower rows of the grid contain the fillet, the remaining are divided between airfoil, edges (first and last column) and top (upper row).


Figure 3.2: On the top left one portion of airfoil obtained with the grid windowing, on the top right the two components distinguished by color. On the lower row the approximation (grid) of a simplified model of the airfoil (green ponts) in the original vertical position and once rotated using the PCA directions.

### 3.1.1 Airfoil partitioning

Each windowed part of the airfoil contain both sides of the surface that have to be separated to allow the parametrization. By using the libraries trimesh [15] and networkx [16] it's possible to build a connectivity graph of the mesh which links together adjacent faces, allowing the distinction of the two sides from each other.

The last step is to translate the mesh to have it centered in the origin and rotate it to help the construction of the approximation.

Those transformations are easy to perform on the airfoil. For each sub-mesh the mean value of its points is found and is subtracted to each of them to center the mesh.

The matrix used to orientate the mesh horizontally is obtained through the PCA. The PCA of the covariance matrix $\Sigma_{n}$ of these translated points is computed through its singular value decomposition $\Sigma_{n}=U S V^{t}$ where the PC direction are the rows of the matrix $U^{t}=\left[d_{i}, d_{2}, d_{3}\right]^{t}$.

The first two components represent the two direction of higher variance, and being the surface almost flat the resulting mesh is almost parallel to the $x y$ plane. The flat structure of the airfoil guarantees that no further adjustment are required. The result can also be used as a
parameterization of the points mapping their $(x, y)$ coordinates into their $z$ ones.
The rotation matrix and the center are stored to reconstruct the damages position at the end of the detection process.

### 3.1.2 Edges partitioning

The edges portions of the mesh are far more delicate because the structure of the edges makes them difficult to approximate with the graph of a function. The two sides of the blade must be separated as for the airfoil but there is also the part that connect them to be considered.

Dividing one side from the other can not be simply done with some kind of connectivity based procedure, given the single piece of mesh involved. There is also no clear distinction between the flat sides and the curved edge except for this qualitative distinction.

Anyway the fundamental issue is that the patches extracted must have a structure simple enough to modeled with splines or polynomials to be properly analyzed.

As a consequence separating the mesh in just two parts would not be enough. At least one of the two sub-meshes would include a portion of the curved edge orthogonal to the flat side part, making it extremely hard to properly model. In fact, once oriented, the thin edge strip would result (almost) vertical and being small in comparison to the whole patch would not influence the approximation sufficiently to be correctly fitted.

A minimum of three sub-meshes are necessary: two for the sides, they will be called borders to distinguish them from the regular airfoil, and at least one for the edge.

Is also important to notice that if the edge portion is too large its structure will became too complex because of the parallel border strips attached. In addition the method used to divide them have to be robust against the deformation that are occasionally present on the edges.

Defining the correct portions to be carved is the hardest part of the partitioning process. Two methods are here exposed to solve this problem. Their results will be shown in the next chapters.

## First edge refining method

The simplest algorithm proposed is based on the triangular faces normal. Chosen a direction orthogonal to one border side as reference the scalar product of it with all the faces normal of the patch is evaluated. If the result is close to 1 is on the same side, if close to -1 on the other, the middle ground is the curved junction part on the actual edge.

The right reference direction and what close means are the parameter of this method and determine where the border parts ends and the curved edge start.

The direction was selected as the mean values of the face normals of a group roughly in the center of one of the flat sides. This group was determined as the neighborhood of faces surrounding the triangle whose barycenter was the closest to the mean of mesh points.

Deciding the minimum (and maximum) values of the scalar product is hard to determine at prior.

The more of the bent part is left on the flatter portion, the hardest it is to model it with a smooth and simple approximant. Even more flexible modeling function would have problem because of the small area on the twisted portion in comparison to the whole piece.

Being otherwise too "strict" is also problematic because the remaining curved part would have two almost parallel strips on the sides. Those parts make a parameterization impossible if left on top of each other (see Fig:2.3). If otherwise rotated to avoid the overlapping the sides would became almost vertical, increasing the difficulty of building a good approximating function, as shown in Fig:2.4a.


Figure 3.3: On the upper row the spline approximation (grid) on a fold similar to the edges (green) with different orientation. On the middle row the spline approximation perdormed on half the fold as it is (left) and with the rotation obtained with th PCA directions (right). The differences of the corresponding approximation error are displayed in the last row and show how such a solution is not sufficient.


Figure 3.4: On the left one portion of the edge obtained with the rough windowing, on the right the two sides in blue and red and the curved edge in green.

Is also possible to leave large portion of the flat sides attached to the edges if the starting reference normal is chosen poorly, or if the surface presents a higher curvature than usual.

Once a good choice of the parameter is found and the three components are build they still have to be oriented in a proper way. The two border parts are treated like in the airfoil case: translated using the center of the mesh and rotated with the matrix of the principal components directions.

The third part require some extra attention. The rotation and translation are both performed but the result could have a much steeper side than the other. This happens because the rotation based on the PCA is influenced by the sizes of the lateral strips, that are usually different. When one is larger than the other that side will be oriented more horizontally given the increased variance.

To better orientate the edge portion a second rotation determined with a brute force method is applied. Working with the points projected along the edge length, their second and third components after the PCA based orientation, rotations with different angles (between $\left[-\frac{\pi}{6}, \frac{\pi}{6}\right]$ ) are applied. A parabola approximant is built for each rotation result and the one which allow the lower error is selected and applied to the original data.

This extra step resulted in a stark reduction of the approximation error. Given the small number of points involved and the simplicity of the approximating function, with only 3 parameters, the time consumption is also conveniently small.

The rotation matrix is then multiplied for the PC direction one and stored in its place to allow for the final reconstruction of the damages positions.

## Second edge refining method

This procedure is more adaptive than the previous one and based on the edge detection algorithm presented in [13]. To start the mesh is roughly divided in two parts based on the faces normal.

For each piece a smaller subset of triangular faces (called building set for convenience) is selected, in the same way as for the first method. The faces' centers of this set are used to build


Figure 3.5: On the left one portion of the edge obtained with the rough windowing, on the right the two sides in blue and green, the two portions of the actual edge in red and yellow.
an approximation with a bi-variate low degree polynomial.
This approximant does not take into account any of the other parts of the portion, and is computationally cheap to perform given the low number of parameters involved and the small set of data used.

The faces of the mesh whose centers are within a certain distance from the approximant become the new building set. The process is then repeated ten times (or until the set start shrinking) to build a collection of faces of the mesh that can be easily fit with a simple approximant, and then should allow a good fit with the spline approximation.

The remaining faces are divided by connectivity and the groups with a small number of points are added to the previous list. This is both to avoid getting a shredded remaining mesh and because the damages in the middle of the flat part are usually separated by the algorithm because they do not fit with the approximation.

The same then apply for the selected faces, in this case only to remove disconnected pieces that could spoil the construction of the spline approximant. The two separated portions are saved as borders and kept separated from the airfoil for reasons that will be cleared in the next chapters.

The faces discarded from the two starting sub-mesh are then joined back together in what is basically the actual curved edge. This residual part is bigger compared to what the first method produced, an ulterior division in two pieces is necessary to simplify its structure.

Using the same approach with two distinct building sets, selected close to the two borders, two patches are identified. The biggest one is saved as first edge portion while the residual, if there is one, as the second one. Trying the construction of a patch on both sides helps in obtaining a bigger patch, useful to reduce the effect of impacts on the approximation, and makes the process less sensitive to the selection of the initial building set.

This method is slower than the first one, but the results are less sensitive to the choice of the reference normal and the patches extracted are inherently easier to approximate, improving the accuracy of the algorithm on the edges.

Each of the sub-meshes obtained is then aligned as with the airfoil and the last two are given
the extra adjustment with the parabola approximant as in the first method. The transformation matrices and centers are stored to recover the original position at the end.

### 3.1.3 Top part

The topmost strip of the grid contains the superior edge of the blade. While the edge was not relevant for the application the transition zone between airfoil and top edge must be analyzed, and require extra care to extract patches regular enough to be approximated correctly.

The same procedure as for the edges is applied. The actual summit is not analyzed further while the two flat parts are stored as "top", separately from borders and airfoil.

### 3.1.4 Corners partitioning

The rapid change in curvature between airfoil and edges of the blade makes the construction of an accurate approximation difficult. In addition to that on the corners the thinness of the blade makes the scanning procedure less reliable, with a result lacking the same smoothness found in the other parts of the mesh.

As shown in the picture in Fig:3.6 the mesh on the edges is jagged and irregular because of measurement errors. The information coming from faces normal become then mostly meaningless and the distinction between the top and side portion of it is made even harder.

The thinness of the corners is also related to a higher chance of bigger deformations and damages caused by chipped off pieces.

The irregularities reduce the usefulness of the approximant as reference model as the corner edge is much more narrow than in the other portions of the edges, and not as straight. The construction of a straightforward approximant or simply orienting it a much more difficult prospect. The flat parts at the sides were stored with the top patches.

Because of these reasons the algorithm discussed in this work was not applied to the corners. A solution to the edge analysis problem will be proposed in the final chapter along with other ideas on how to further improve and modify the algorithm.

### 3.1.5 Fillet partitioning

The first step to divide the fillet is to separate the points belonging to the horizontal base from the points belonging to the junction (and the blade itself). To do that the normals are once again used: all the triangles with a normal (almost) parallel to the vector orthogonal to the lowest horizontal line of the grid belong to the base, the remaining to the fillet.

The fillet portion can then be partitioned in the same way as the rest of the blade: for every horizontal strip the first and last cell of the grid are to be treated as edges and the other as the airfoil.

The main difference between the fillet and the other parts is the higher curvature of the surface. This doesn't have any effect on the separation of the two sides for the airfoil cells, but makes the separation of the edges much more sensitive to the initial choice of reference normal or building set as shown in Fig:3.8c.

The first method for the edges refinement is not effective and only the second showed acceptable results, even if the junction part at the base is often divided in a less regular way than expected. As an example, Fig:3.8d shows how the red patch is not divided with a straight line from the blue border piece. This level of irregularities did not caused difficulties for the approximation, while the situation in Fig:3.8c proved to be much harder to fit correctly.


Figure 3.6: The edge of a corner portion (extracted with the first method) represented as a surface, made of the triangular faces of the mesh.

### 3.2 Approximating function choice

To build an artificial model of the surface windows the family of functions used must be capable of giving an accurate approximation of the cloud of points but smooth enough to avoid fitting the impacts. It should also be identifiable in reasonably low times to guarantee a good usability.

Polynomyals and splines functions have been used for this purpose because of their flexibility and thanks to the availability of libraries to efficiently work with them, like splipy [17].

The flexibility of a polynomial approximant can be tuned changing the degree along each axis. Keeping the degree low the result will be smoother, less adaptable and so ignore the irregularities more easily. The low number of parameters to be found will also fasten the computations required for its evaluation and initial identification. Increasing the degrees reduces the approximation error and improve the fitting but might overfit and also model the damages.

As an example, the patches on the airfoil can be modeled correctly with five by five degrees polynomials, that need only 15 parameters per window to be defined. On the other hand working with the edges showed that the more complex geometries require higher degrees to keep the residual error low enough to perform further analysis, increasing however the risk of fitting the damages in the model.

Spline approximants, as described in (2.3), allow a greater adaptability as the number of control points grow. The structure of the splines is also different from the one of the polynomials.

Polynomials are global approximants, in the sense that each parameter that defines them influence every point in the domain, with the exception of the axis at best. Splines are instead a linear combination of B-spline functions, each of which has support in only a small portion of the domain. This implies that modifying the value of a control point will cause local modification of the spline, leaving the portion outside of the respective B-spline function domain unaltered. This allow a better approximation of the mesh, especially on the edges where sharp turn and steep inclines are present, but increase substantially the risk of modeling the damages. To lessen


Figure 3.7: Fillet portion of the mesh, after the removal of the base.

(a) Detail of the fillet edge.

(c) Example of bad partitioning.

(b) Detail of fillet airfoil.

(d) Example of correct partitioning.

Figure 3.8: In Fig:3.8c and Fig:3.8d the border points are represented in green and blue while red and yellow are the first and second portion of the edge. The irregular structure of the edge portion in Fig:3.8c is caused by the incorrect selection of the initial building set.
this problem a smoothing parameter can be added to the formulation as described in (2.3.1).
The cost of evaluating a spline function also is higher than the one for polynomials because of the underlying cost of evaluating separately the basis functions on each input point. This limitation depends on the library used and not on the splines directly but it is important to remember it when comparing the time results in the next chapter.

Must be also remembered that spline functions have a domain defined from their knot position instead of the data. To uniform the process the domain was kept at $[0,1]^{2} \subset \mathbb{R}^{2}$ for every patch and a transformation applied to the input points to move the corner of the window to the corner of the unit square.

Once build, the approximant was evaluated on the $x y$ coordinate of the input points and those values used to compute the differences necessary to locate the damages.

### 3.3 Strategy for identifying potential location of the damages

The identification of the damage locations and the portions of the patches involved require the comparison between data and the artificial model that is the approximation. The absolute values of the differences between the points in the portion being analyzed and the corresponding points on the parametric approximant are the instrument necessary to do it.

Selecting the correct points for each damage is fundamental to perform a correct study on their shapes, highlighting the importance of the approximating model to obtain meaningful values for the differences

A critical part for the difference evaluation are steep slopes. Even if the distance between model and data is small, defined as minimum distance between a point of the mesh and the points of the approximant, the difference between their third components can be higher. Controlling this aspect should be done in the partitioning part of the algorithm by dividing and re-orienting zones with a steep incline.

The evaluated differences have to be filtered with a minimum threshold, chosen so that the unavoidable measurement noise and the blemishes too little to be relevant can be ignored. Every value under the threshold is rounded to zero. This parameter will be called first parameter or first threshold in the results chapters.

After the filtering is performed the points with positive difference values are selected and must be separated between the differnt damages.

To distinguish different locations the solution found was to build a sub-mesh with only the triangular faces marked by the high difference and build a connectivity graph of them. Using the connectivity graph is then easy to distinguish separated cluster of points, in the same way as with the two sides of the airfoil for the portioning.

To avoid taking too few points to perform meaningful further analysis the location considered are enlarged by two layer of neighbor triangles or, in case of single points, by five layers of neighbors. This affect the size of the marked zones, but the filter always remove some points at the base of the damages so this steps mainly compensate that as saw in figure 2.5 .

Each cluster of points is then analyzed independently.

### 3.4 Damage classification

Once each zone is isolated, they are translated so that their center is in the origin of the axis. At this point PCA can be performed on the covariance matrix of the points in it, but with the differences evaluated previously in place of the $z$ values. This is necessary to reduce the effect of the surface curvature in the analysis.

What must be measured is the entity of the deformation, the divergence from the pristine form, and not the shape of the points itself, that is dependent on the surface. It is intuitive how a scratch on a plane and the same scratch on a curved surface are different, the same holds for impacts. Using the differences enable us to ignore the shape of the mesh to a certain degree and work only with the variation induced from the damage.

The relative sizes of the impacts in this application guarantee that the third component, the smallest one, is always the depth, because length and width are always bigger. The location marked are analyzed to determine their real dimensions. To do this the cluster of point is reoriented by using the principal component direction as in the portioning part, then length, width and depth can be measured along the axis as the maximum differences along the new axes.

The principal components values are then normalized and the location classified based on the results. Normalizing the values is done to work with the relative magnitude of the PC values instead of their values. The structure of the impacts usually present values with the same order of magnitude of $10^{-1}$ for the first and second normalized PC values while the third was roughly $10^{-4}$.

The normalization is necessary because the variance along the PC directions can change substantially depending on the number of points in the damage area and their position, making a direct comparison difficult. In addition this solution is similar to what proposed in 2.23, in fact the normalized third PC value, the sphericity and total curvature parameter were usually extremely similar, making the use of all of them only redundant.

Using the correlation matrix instead of the covariance one is another way to reduce the effect of the variance. However the result obtained from the correlation matrix, while having the same directions, were not helpful to understand the damages shapes. The PC values were in fact almost identical between each other, making their comparison useless, because the division with the variance and no recognizable pattern was found.

This difference is much bigger than the actual difference between depth and let's say width, which is usually of only one or two orders of magnitude. The reason behind it is that the points are not analyzed directly, instead the covariance matrix is used. The definition of the covariance includes the multiplication of the (normalized) data with themselves, effectively resulting in rising to the square all the values. The square roots of the PC values show the correct ratio.

To select the impacts from all the damages the first two components are initially compared, filtering out the cases where the second is much smaller compared to the first. By imposing a minimum to the third components the resulting one are then confirmed or discarded. Depth and width are also checked to filter out damages too shallow or thin, but can only be used as auxiliary parameters given the variety of shapes the impacts appear into.

It is also possible to evaluate a reasonable approximation of area and volume of the damages by building the convex hull of the $x y$ coordinates and the whole coordinates (with differences on the third coordinate) respectively.

### 3.5 Components analyzed

As already stated, the aim of this work is to define a procedure to locate and measure the FODs on inner components of jet engines, in addition the algorithm developed should be applicable to different typologies of blades. The performance of the method is going to be shown in the next two chapter on two distinct cases, focusing on different aspects of the algorithm.

In chapter four scans of newly manufactured blisks are analyzed looking for manufacturing defects, testing the capability of the algorithm to detect any kind of damage. In chapter five instead the algorithm is applied to scans of worn compressor engine blades to determine also the efficacy of the classification part of the method.


Figure 3.9: On the left a photo of a blisk, where blades and disk are one single piece, on the right an example of separate blades mounted to the disk.

A blisk, or bladed disk, is a single component of turboengines comprehensive of both rotor disk and blades [18]. The analysis is still going to be performed for a single blade at time. The scans structure is not much different from what shown for single blades beforehand and the shape is also similar, requiring only a slight adjustment of the initial partitioning grid.

The main reason to show this specific application is the complete lack of corrosion on the pieces, because the scans are done right after the production as a mean to check the quality of the objects. The control is performed visually in this case too at the moment. The condition of the objects removes the necessity of a final distinction of the results based on the source. The imperfections to be found are also bigger and fewer in number compared to the one on used blades.

The second set of results will involve blade scans from high pressure compressor jet engines, measured after their use. Those components show a mixture of corrosion and impacts, in addition to the deformations discussed beforehand. This will test the robustness of the method to bends and local deformations, and the performance of the method in distinguishing FODs and corrosion.

In both cases the results are distinguished based on the portion involved: airfoil, borders, edges, and for the blades also topmost strip. The extremely low number of damages on the fillet does not allow for the result to be compared meaningfully to the other portions and have been omitted.

The division of the results helps in determining which characteristics of the surface are more important for the correct functioning of the algorithm. The portions where further research is required and the reason of such limitations will also be discussed.

### 3.6 Result structure

The tests performed aim to show the effect of three main parameters on the analysis:

- flexibility of the approximating function,
- threshold for the difference filtering,
- threshold for the third value of the normalized PC values.

The results of the algorithm set with combinations of those parameters will show their influence on the process and indicate the strategy do follow to select the optimal ones.

While the three elements affect different parts of the method it is not possible to analyze them separately. The only way to analyze their effects is to compare the number of missed damages, false positives and correct findings. Specifically the following quantity will be used:

$$
\begin{align*}
\text { Completeness } & =\frac{T P}{T P+F N} \\
\text { Correctness } & =\frac{T P}{T P+F P} \tag{3.1}
\end{align*}
$$

where $T P$ represent the number of true damages to be found, $F N$ the number of damages not detected and $F P$ is the number of location marked incorrectly as damages. Completeness then represent the percentage of the total damages detected and correctness the percentage of the results pointing to an existing damage.

The true positives were defined as the one marked as damages through visual analysis performed independently from this work.

### 3.6.1 Approximant parameters choice

As stated beforehand, two family of functions are used as approximants: polynomials and splines. In the case of polynomials modifying their degree along each axis changes their degrees of freedom. For splines the degrees was fixed at two ${ }^{1}$ and the number of control points was modified to change the flexibility of the approximating functions.

The position of the control points was determined using the same method as their values with 2.4 but with the input points $x$ and $y$ components of the points as $z_{k}$. This way the distribution of control point is similar to the distribution of points of the mesh.

This resulted in almost uniformly equispaced control points in the majority of cases, but (marginally) improved the results in patches such the edges. On those the mesh points are almost equispaced on the surface while are not uniform in the reference system used, being it a projection on the plane formed by the first two PC directions.

In the tests the values $[3 \times 3,5 \times 5,10 \times 10,20 \times 20]$ were used for both families of functions, to allow for a comparison. To simplify the notation a spline function with a grid of $n \times n$ control points will be referred to as a $n \times n$ spline, as polynomials of degrees $n \times n$ will simply be called $n \mathrm{x} n$ polynomials.

It has to be noticed that while the parameters of splines are the control points, so a $n \times n$ spline has $3 n^{2}$ parameters to be defined i.e. the coordinates of all the control points, a $n \mathrm{x} n$ polynomial has $\frac{n(n+1)}{2}$ parameters. As a consequence splines always have more parameters and higher flexibility than polynomials but it also means a higher computational cost.

Increasing the parameters number will increase the model adaptability and decreasing the error, but also increasing the risk of fitting the damages and making their detection harder or impossible. An optimal value is then one such that allow with enough accuracy to model only the structure of the mesh and nothing else.

The quality of the fit then can not only be measured by the residual error

$$
\text { res }=\sum_{i=0}^{N}\left(f\left(x_{i}, y_{i}\right)-z_{i}\right)^{2}
$$

[^2]or the root mean square error
$$
R M S E=\sqrt{\frac{\sum_{i=0}^{N}\left(f\left(x_{i}, y_{i}\right)-z_{i}\right)^{2}}{N}}
$$
between data $\left(x_{i}, y_{i}, z_{i}\right)$, for $i=1, \ldots, N$ with $N$ number of points, and $f$ being the model, but as the residual on only the undamaged parts.

This can not be done prior to the detection of the damages obviously, the quality of the approximation has to be checked together with the quality of the analysis.

### 3.6.2 Thresholds choice

The difference values between the approximation and the original data must be filtered to remove noise and as much corrosion damage as possible. The minimum value chosen for this selection will be referred to as first parameter or first threshold from now on.

The first parameter considered will vary between $[0.003,0.005,0.010,0.020]$ and were determined heuristically as a good choice for the application. Those are dependent on the size of the damages to be found and the noise level.

Low values for the first parameter mean analyzing bigger portions of the mesh allowing for the detection of small imperfection. Higher values will instead exclude the smaller differences analyzing only the bigger variations.

The optimum value for this parameter should remove only the approximation error and keep only the damages. The minimum depth of a relevant impact damage is 0.02 mm , using lower values guarantees that the slopes of the impacts do not get excludes from the analysis.

This parameter also influences the computational time because it changes the number of points to be analyzed. The smaller the parameter, the more points to be checked, the slower the process. This effect is usually minor compared to the cost of building and evaluating the approximating functions.

The choice of the minimum value of the third value of the normalized PC values determines the shape, the roundness, of the damages marked as impacts. Its value will be referred to as second parameter or second threshold.

The heuristically determined thresholds used in the tests varies between [0.0001, 0.0002 , $0.0005,0.0010]$. Lower values for the second parameter allow shallower damages to be accepted, higher values consider only the imperfection with rounder and deeper shapes.

The second parameter does not affect the speed of the algorithm, but is the main criteria to distinguish between corrosion and actual impacts.

### 3.6.3 Displayed results

The parts of blades tested are analyzed separately, depending on their structure: airfoil, edges, borders top. The results are displayed separately for splines and polynomials, in addition to a table comparing their residual error and RMSE.

Comparing the results between different portions of the blades gives information on the behavior of the algorithm in presence of different surface shapes. Making a comparison between splines and polynomials findings is instead useful to understand which kind of approximation works better on each part.

The details of the analysis are shown with four graphs and a table, as in Fig:4.2 and table 4.2 .

The graphs show completeness, correctness and computational time for each combination of the first and second parameters and RMSE for the single patches checked. In every graph the

Parameters combinations

| Second parameter | First parameter |  |  |  |
| :--- | :---: | :---: | :---: | :---: |
|  | 0.003 | 0.005 | 0.010 | 0.020 |
| 0.0001 | 0 | 4 | 8 | 12 |
| 0.0002 | 1 | 5 | 9 | 13 |
| 0.0005 | 2 | 6 | 10 | 14 |
| 0.0010 | 3 | 7 | 11 | 15 |

Table 3.1: Table explaining the parameters combinations displayed in the results graphs. As an example, the values in the sixth position is the result for a first threshold value of 0.005 and a second threshold of 0.0005 .
approximants are distinguished with different colors based on their degrees or number of control points, as described in the legend.

The first three graphs of each figure are divided by vertical lines to ease the distinction between the parameter used. Each interval of four points, starting from zero, present the results for a single value of the first parameter while the second iterate between its four values. Table 3.1 display the parameters combinations related to each position on the axis.

When no damages where found the correctness value was marked as Nan, and the corresponding point in the correctness graph is omitted.

The RMSE graph is instead indexed by the number of patches analyzed. To understand what part of the mesh they refer to is necessary to understand how the sub mesh obtained from the partitioning are ordered.

The patches of the airfoil are extracted by the partitioning process one row at a time, starting from the bottom. Each row is then divided in cells and the cells are refined (in at least two parts) from the left to the right with respect to Fig:3.1a. The first patch is then the one on the lower left corner, increasing the index moves to the right until the end of the row then restart on the left of the superior row.

Border and edge patches are partitioned from the bottom to the top but the left and right edges are treated separately and then joined. The first half of the patches are then the one from the left edge and the second half come from the right one, in both cases going from the lower to the upper cells.

The tables display numerically the best and worst results obtained for each approximant tested, sorting for decreasing order of importance completeness, correctness and computational time. When all is written it means no significant variation in the results changing that parameter.

## Chapter 4

## Application of the detection algorithm to blisk analysis

The algorithm proposed is designed for the detection of small impacts on blades that have already flown. A second natural application of the same procedure is to check newly manufactured components to detect imperfection caused by the production process.

This chapter shows the result in the latter case, analyzing newly manufactured blisks blades looking for any kind of imperfection involved.

Blisks, term obtained by the union of BLade and dISK, are engine components where blades and rotor disks built in a single piece. Blisk blades have different mechanical properties and associated costs compared to single blades, refer for example to [18] for additional detail on the advantages and drawbacks involved.

In particular their maintenance cost is usually more expensive than with single blades, being them the union of multiple part. Extensive and accurate quality controls are then necessary for the newly manufactured blisk to guarantee the maximum integrity of the object.

The application of the proposed method to detect surface damages on the blade parts of the blisk can help in this sense. The same data structure and similar shape of both blades and damages allow the algorithm to work on both with basically no modification on the procedure.

About the structure of the blades tested the main differences between regular blades and blisk's blades are in their superior parts. In blisk's blades the upper part has a different lateral inclination compared to the lower one. The blade requires a different structure for the partitioning grid, but once the six corner points are defined they can be used for all the blades on the blisk. This uniformity depends on the lack of any deformation on the structure of newly produced components and constitute an advantage compared to the application presented in the next chapter.

As the figure Fig:4.1 shows, the superior portion has an extra curve absent in the lower one. This add complexity to the surface structure in the top portion.

There is also a difference in the damages structure, apart from the lack of corrosion. While in the FOD detection we look only for indentation on the surface, on the newly built blisks extruding parts are also searched. The way damages are found, through the difference between approximation and original data, can easily ignore the direction of anomalies by just using the absolute values of such differences.

Differentiating bulging parts from indents is in fact slightly harder and requires taking into consideration the orientation of the mesh given by the normals of the mesh faces. Comparing the normals orientation and the differences signs is necessary because the approximating function is usually oscillating between both sides of the mesh.

It must be noted that the number of damages on the newly produced pieces is low, in


Figure 4.1: On the left the analyzed part of the blisk blade, on the superior portion of it is visible the extra curve. On the right the grid for the first step or the windowing.

Airfoil approximation error

| Polynomials | $3 \times 3$ | $5 \times 5$ | $10 \times 10$ | $20 \times 20$ |
| :--- | :---: | :---: | :---: | :---: |
| res | 0.8661 | 0.2943 | 0.1948 | 0.1226 |
| RMSE | $2.9280 \mathrm{e}-6$ | $9.9512 \mathrm{e}-7$ | $6.5873 \mathrm{e}-7$ | $4.1453 \mathrm{e}-7$ |
| ctrl pts | $3 \times 3$ | $5 \times 5$ | 10 x 10 | 20 x 20 |
| res | 2.1134 | 0.4643 | 0.2095 | 0.1191 |
| RMSE | $7.1450 \mathrm{e}-6$ | $1.5697 \mathrm{e}-6$ | $7.0853 \mathrm{e}-7$ | $4.0282 \mathrm{e}-7$ |

Table 4.1: Residual error and root mean square error for polynomial and splines approximating functions.
particular no imperfections were found in the top portion or on the fillet for the blades tested, so only airfoil and edges will be discussed in this chapter.

The same set of parameters will be used and displayed in all the following analysis to allow for a better comparison of the results. A description on their formulation is provided in section 3.6 .

### 4.1 Airfoil

The airfoil represents the biggest and flattest part of the blade. This flatness makes orienting the surface extremely easy and using the PC direction to do it is effective.

This happens because the variance is mainly spread in two directions, basically the same ones that we would obtain approximating the surface with a plane. The preparation for the construction of the approximant is then straightforward, without the problems related to vertical parts in the mesh.

In this first analysis polynomial and spline results are discussed separately for more clarity.

Airfoil, Polynomials


Figure 4.2: In figures 4.2a 4.2b 4.2c completeness, correctness and computational time for polynomial analysis for the sets of parameters tested. The vertical lines divide the graphs in intervals $[a, b)$ based on different values of the first parameter. Figure 4.2d the RMSE is shown for each patch analyzed. The legend displays the degrees of the polynomials used.

| degrees | $3 \times 3$ |  | $5 \times 5$ |  | $10 \times 10$ |  | $20 \times 20$ |  |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | Best | Worst | Best | Worst | Best | Worst | Best | Worst |
| diff filter | 0.010 | 0.020 | 0.010 | 0.020 | 0.010 | 0.020 | 0.003 | 0.020 |
| PC value | 0.0010 | 0.0001 | 0.0010 | 0.0010 | 0.0005 | 0.0005 | 0.0010 | 0.0001 |
| time | 2.19 | 0.98 | 1.88 | 1.73 | 6.80 | 6.83 | 34.73 | 34.51 |
| compl | $100 \%$ | $50 \%$ | $100 \%$ | $50 \%$ | $100 \%$ | $50 \%$ | $100 \%$ | $0 \%$ |
| corr | $100 \%$ | $100 \%$ | $100 \%$ | $100 \%$ | $100 \%$ | $100 \%$ | $66 \%$ | Nan |

Table 4.2: Summary of the best and worst results obtained from the tests on the set described in 3.6.

Polynomial results. In Fig: 4.2 d the RMSE is shown for each sub-mesh in the airfoil and the error values highlight how some patches are harder to model. The first and last groups of meshes of the sequence are respectively the one on the lower and higher part of the blades. Those parts are characterized by a higher curvature of the surface. The upper part present a curve as shown in Fig: 4.1, the lower strip is close to the curved fillet and start changing shape accordingly.

The $3 \times 3$ polynomial presents a consistent increase in the RMSE there, the values were reduced increasing the degrees of the approximants. Two other groups of meshes, roughly around the 60 th and 160 th, show higher errors even increasing the degrees.

The reason for the reduction of the error is the increase in flexibility of the approximant that became capable of fitting the more complex geometry of those patches. For the curved ones this is particularly visible since the error reduces significantly between the 3 x 3 and 5 x 5 setup.

The meshes where instead the error does not decrease are the one where damages are located. The presence of the damage means greater differences between model and data even if higher degrees of the polynomials partially reduce it. However it is better to use a low degree approximant in order to be able to detect all damages.

In Fig: 4.2 a the damages are found for each choice of the polynomial degree for at least some of the values of the parameters.

The $3 x 3$ polynomials show a good performance in terms of completeness and the correctness is usually good. The computational time required changes visibly depending on the first parameter instead.

This happens because the approximation is sufficient to capture the general shape of the mesh but it is still inaccurate and the initial first parameter value is too low to properly remove the approximation error. This lead to a high number of mesh points marked as potential damages and analyzed only to be discarded later. Then, it can be seen how the increasing in this threshold level lowers the computational cost, without affecting completeness and correctness.

The best results are obtained with $5 \times 5$ polynomials, in both time cost and correctness. The approximation error lowers considerably with respect to the $3 \times 3$ case, mainly in the first and last patches. Looking at table 4.1, it can be seen how the residual error is roughly one third than for the $3 \times 3$ polynomial. The lack of changes in computational time hint that the approximation error (outside the damages) is lower than the lowest first parameter.

Increasing the degree to $10 \times 10$ lower further the residual error and RMSE but almost quadruples the computational time. This is caused by the increase in cost to define the bigger pool of parameters of the polynomial, going from 15 to 55 . The improvement in general accuracy is also only a fifth of the gain obtained going from $3 \times 3$ to $5 \times 5$. This means a lowering cost-efficacy ratio and an increased risk of overfitting.

The worst results in completeness and correctness are obtained with $20 \times 20$ degrees. In addition the computational time is almost eighteen times the one for $5 \times 5$. The lowered accuracy of the detection is due to both overfitting of the big damages and small oscillation in the approximating function.

The correctness graph in Fig: 4.2b, when looked on the single intervals, display increases in the values.

The effect of the second parameter is what causes this behaviour. By restricting the requirements for an imperfection to be classified as damage, false positives are reduced. ${ }^{1}$ This is strongly related to the shape and dimension of the damages deemed relevant.

This shows that $5 \times 5$ degree for polynomials, low values for the first parameter and high values for the second one form the best settings to analyze this part of the surface. This can easily

[^3]be explained with the combination of a simple structure, resembling a saddle point, and the necessity to leave as little leeway as possible to the approximant to avoid modeling the damages.

Splines results. The results obtained from the splines are similar to the ones from polynomials in terms of completeness, while correctness is generally lower. Here too low values of the first parameter give the highest number of correct findings in general.

Fig:4.3d shows higher approximation errors on the lower and higher portions of the grid, where the surface has extra curves, compared to the middle one like with polynomials, but with higher error in general. This is especially evident for $3 \times 3$ splines, and increases in the number of control points, even only to $5 \times 5$, lower the RMSE sensibly in those parts. The same smaller but stable spikes are present on the meshes with the damages.

The time graph 4.3 c shows a huge increase in the computational cost necessary to perform the analysis compared to the polynomial one. The time required is, at minimum, seven times higher than with the slowest polynomial. Here too the first parameter distinctly influences the time cost for the $3 \times 3$ spline.

Those observations combined suggest that $3 \times 3$ spline approximant is insufficient to properly approximate the surfaces analyzed. The reasons are the same as with polynomials, only slightly more evident. Is also easy to see that a grid of $5 \times 5$ control points gives the best results in terms of time, completeness and correctness, as the corresponding polynomial.

Increasing the flexibility of the approximant with $10 \times 10$ control points decreases the approximation error but does not improve the performance of the algorithm, instead slows it down and makes it more sensitive to the second parameter choice.

Raising the number of control points to $20 \times 20$ makes the approximating function less useful as reference model, causing more false positives to appear.

We can conclude that a low number of control points is the best choice for the approximating splines, allowing the fitting of the simple structure of the airfoil without modeling the damages. Modifying the values for the thresholds can improve the results, like with the $3 \times 3$ case, but can't do much if the artificial model is also fitting the damages or oscillating like for the 20 x 20 splines.

Splines and polynomials do not show substantial differences when used with the correct set of parameters. This suggest that, at least on the airfoil, polynomials are the best solution given the lower computational time needed.

### 4.2 Edges

While the choice of how to partition and orientate the patches on the airfoil is straightforward, the same can not be said of the edges.

Deciding where the edge ends and the airfoil starts is debatable since the mesh we work with come from a physical object and not from some purely virtual model. Small variations caused by measurement and manufacturing process are unavoidable and make any a priori decision on its position unreliable.

However a correct differentiation between the two parts is extremely important to simplify as much as possible the structure of the patches. This step guarantee that the approximation can be performed with as few degrees of freedom as possible, reducing the risk of fitting the damages.

The two methods proposed to partition the edges deal with the problem in two different ways. The first and more straightforward one uses the orientation of the triangular faces with respect to a reference direction to select the two airfoil sides. The second method expands from a starting set on each side until the approximation does not work anymore and then repeats for the remaining parts.


Figure 4.3: In figures 4.3a 4.3b 4.3c completeness, correctness and computational time for splines analysis for the sets of parameters tested. The vertical lines divide different values of the first parameter. Figure 4.3d the RMSE is shown for each patch analyzed. The legend displays the number of control points used.

| ctrl pt | $3 \times 3$ |  | $5 \times 5$ |  | $10 \times 10$ |  | $20 \times 20$ |  |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | Best | Worst | Best | Worst | Best | Worst | Best | Worst |
| diff filter | 0.005 | 0.010 | 0.005 | 0.010 | 0.003 | 0.020 | 0.005 | 0.020 |
| PC value | 0.0001 | 0.0001 | 0.0002 | 0.0002 | 0.0010 | 0.0001 | 0.0005 | 0.0005 |
| time | 91.47 | 82.20 | 78.33 | 76.92 | 97.18 | 99.75 | 223.72 | 229.01 |
| compl | $100 \%$ | $50 \%$ | $100 \%$ | $50 \%$ | $100 \%$ | $0 \%$ | $100 \%$ | $0 \%$ |
| corr | $100 \%$ | $50 \%$ | $100 \%$ | $100 \%$ | $100 \%$ | nan | $50 \%$ | Nan |

Table 4.3: Summary of the best and worst results obtained from the tests on the set described in 3.6.

Border approximation error, first method

| Polynomials | $3 \times 3$ | $5 \times 5$ | $10 \times 10$ | $20 \times 20$ |
| :--- | :---: | :---: | :---: | :---: |
| res | 0.4612 | 0.1405 | 0.0468 | 0.0168 |
| RMSE | $1.1393 \mathrm{e}-5$ | $3.4721 \mathrm{e}-6$ | $1.1562 \mathrm{e}-6$ | $4.1577 \mathrm{e}-7$ |
| ctrl pts | $3 \times 3$ | $5 \times 5$ | $10 \times 10$ | $20 \times 20$ |
| res | 1.2508 | 0.2514 | 0.0719 | 0.0260 |
| RMSE | $3.0895 \mathrm{e}-5$ | $6.2103 \mathrm{e}-6$ | $1.7760 \mathrm{e}-6$ | $6.4266 \mathrm{e}-7$ |

Table 4.4: Residual error and root mean square error for polynomial and splines approximating functions.

To distinguish the airfoil parts close to the edge from the regular airfoil they are called border pieces and they will be considered separately. The reason is the different shape of the surface near the edge, that require a higher adaptability of the approximating function to be properly modeled.

The remaining part will be called edge and studied separately because of its shape resembling a cylindrical paraboloid instead of the almost flat airfoil.

The same kind of analysis performed for the airfoil is shown for the patches resulting from the two methods, for both borders and edges. The damages to be detected were on the dividing line, making their position detectable in both border's and edge's patches.

### 4.2.1 First partitioning method

The following results were obtained using the first method described in section 3.1.2 with a tolerance of 0.95 to distinguish borders and edges. This means that once a reference direction $v=\left(v_{1}, v_{2}, v_{3}\right)$ of unit length is selected all the triangular faces of the mesh with a unit normal $n=\left(n_{1}, n_{2}, n_{3}\right)$ such that

$$
v \cdot n=\left(v_{1} * n_{1}+v_{2} * n_{2}+v_{3} * n_{3}\right) \geq 0.95
$$

belong to the one of the sides of the border, if $v \cdot n \leq 0.95$ then is part of the second. The values in the interval $(-0.95,0.95)$ are assigned to the edge. This value was heuristically determined, and performed correctly in the majority of cases.

The resulting border pieces are then composed of triangular faces almost parallel to each other, this means achieving a certain flatness.

It is easy to imagine how a bad choice of the reference direction might negatively affect the subdivision of the mesh. To determine a good one we started from the point that was the closest to the mean value of the mesh vertexes. The mean value of the triangular face normals in a small neighborhood surrounding the point is then used as reference.

The superior parts of the blisk blades, where there is an additional curvature on the surface, interfered with the partitioning.

Requiring only the selection of the reference direction and the computation of the scalar product on each patch this method is fast, taking only 3 seconds.

## Border

In tables 4.5 and 4.6 and in Fig:4.4,4.5 the results of the algorithm on the border obtained with the first method are displayed.

Border, Polynomials, first method


Figure 4.4: In figures 4.4 a 4.4 b 4.4 c completeness, correctness and computational time for polynomial analysis for the sets of parameters tested. The vertical lines divide the graphs in intervals [a,b) based on different values of the first parameter. Figure 4.4 d the RMSE is shown for each patch analyzed. The legend displays the degrees of the polynomials used.

| degrees | 3 x 3 |  | $5 \times 5$ |  | 10 x 10 |  | 20 x 20 |  |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | Best | Worst | Best | Worst | Best | Worst | Best | Worst |
| diff filter | 0.010 | 0.003 | 0.010 | 0.020 | 0.005 | 0.003 | all | all |
| PC value | 0.0005 | 0.0005 | all | all | all | 0.0002 | all | all |
| time | 0.67 | 4.95 | 0.32 | 0.28 | 1.02 | 1.14 | 5.32 | 5.45 |
| compl | $100 \%$ | $0 \%$ | $100 \%$ | $0 \%$ | $50 \%$ | $0 \%$ | $0 \%$ | $0 \%$ |
| corr | $100 \%$ | $0 \%$ | $100 \%$ | Nan | $100 \%$ | $0 \%$ | Nan | Nan |

Table 4.5: Summary of the best and worst results obtained from the tests on the set described in 3.6.

Border, Spline, First method


Figure 4.5: In figures 4.5 a 4.5 b 4.5 c completeness, correctness and computational time for splines analysis for the sets of parameters tested. The vertical lines divide different values of the first parameter. Figure 4.5 d the RMSE is shown for each patch analyzed. The legend displays the number of control point used.

| ctrl pt | $3 \times 3$ |  | $5 \times 5$ |  | $10 \times 10$ |  | $20 \times 20$ |  |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | Best | Worst | Best | Worst | Best | Worst | Best | Worst |
| diff filter | 0.005 | 0.003 | 0.010 | 0.003 | all | all | all | all |
| PC value | 0.0005 | 0.0005 | 0.0005 | 0.0010 | all | all | all | all |
| time | 27.32 | 26.80 | 17.31 | 21.63 | 19.75 | 18.65 | 32.19 | 32.82 |
| compl | $100 \%$ | $0 \%$ | $100 \%$ | $0 \%$ | $0 \%$ | $0 \%$ | $0 \%$ | $0 \%$ |
| corr | $100 \%$ | $0 \%$ | $100 \%$ | $0 \%$ | Nan | 0 | Nan | Nan |

Table 4.6: Summary of the best and worst results obtained from the tests on the set described in 3.6.

# Edge approximation error, first method 

| Polynomials | $3 \times 3$ | $5 \times 5$ | $10 \times 10$ | $20 \times 20$ |
| :--- | :---: | :---: | :---: | :---: |
| res | 1.5455 | 1.3567 | 1.2589 | 1.2182 |
| RMSE | $6.7811 \mathrm{e}-5$ | $5.9527 \mathrm{e}-5$ | $5.5235 \mathrm{e}-5$ | $5.3449 \mathrm{e}-5$ |
| ctrl pts | $3 \times 3$ | $5 \times 5$ | $10 \times 10$ | $20 \times 20$ |
| res | 1.8966 | 1.4389 | 1.4974 | 1.4921 |
| RMSE | $8.3213 \mathrm{e}-5$ | $6.3134 \mathrm{e}-5$ | $6.5699 \mathrm{e}-5$ | $6.5469 \mathrm{e}-5$ |

Table 4.7: Residual error and root mean square error for polynomial and splines approximating functions.

Looking at figures 4.4 d and 4.5 d and comparing them to 4.2 d and 4.3 d we can see how the RMSE is higher on the border. In particular one value stand out from the other, corresponding to patch close to one of the corners.

The table 4.4 also shows a higher residual errors with respect to table 4.1 , once considered that the number of patches in the border is less than a quarter of the one in the airfoil.

The values of both completeness and correctness are generally lower than the one obtained for the airfoil. The first parameter associated with the best results is 0.010 , while for the airfoil lower values it performed better.

The number of patches involved is roughly a quarter of the one for the airfoil and the time necessary to perform the analysis is generally proportional to that. One exception is the 5 x 5 case, for both polynomials and splines, where the time necessary shows variations depending on the first parameter while it was more stable on the airfoil. The 3 x 3 approximant computational time is also higher than a quarter of the corresponding values for the airfoil.

The reason behind the different error levels is the lack of accuracy of the approximation concentrated near the edges, where the damages are also located. The fast change in curvature is not properly modeled by the approximation increasing the error locally.

Good results are obtained mainly with higher first threshold values to compensate for it. The $5 \times 5$ approximants show the best results, but the only sets of parameters that gives optimal results for both splines and polynomials are the one where the first threshold is set at 0.01 .

The error levels also affect the computation time increasing substantially the area to be analyzed. Raising the first parameter reduces the time for both 3 x 3 and 5 x 5 cases, even halving it for the $3 x 3$ polynomials.

The analysis time for 10 x 10 and 20 x 20 approximants are mostly unaffected by the first parameter. This means that the error levels are already small enough to be filtered with the lowest first threshold. On the other hand polynomials of 10 x 10 degrees offers mediocre results at best in term of findings, and the correspondent splines can not detect any damage correctly.

Further increases of the approximation flexibility result in the modeling of the damages. In fact no imperfection are located using $20 \times 20$ polynomials and only false positives are returned from $20 \times 20$ splines.

The result shows that even on the border 5 x 5 degrees polynomials in those parts are the best choice, requiring however a more restrictive filtering on the differences between model and data. The same applies to splines where $5 \times 5$ grids of control points allow accurate findings, with the same condition on the first parameter, but requiring more than 20 times the computational time of the respective polynomial.

Edge In tables 4.8 and 4.9 and in Fig:4.6 4.7 we can see the results of the algorithm on the edges portions obtained with the first partitioning method with polynomials and splines respectively.

Edge, Polynomials, First method


Figure 4.6: In figures 4.6 a 4.6 b 4.6 c completeness, correctness and computational time for polynomial analysis for the sets of parameters tested. The vertical lines divide the graphs in intervals $[\mathrm{a}, \mathrm{b})$ based on different values of the first parameter. Figure 4.6 d the RMSE is shown for each patch analyzed. The legend displays the degrees of the polynomials used.

| degrees | $3 \times 3$ |  | $5 \times 5$ |  | $10 \times 10$ |  | $20 \times 20$ |  |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | Best | Worst | Best | Worst | Best | Worst | Best | Worst |
| diff filter | 0.005 | 0.003 | 0.005 | 0.003 | 0.005 | 0.020 | 0.003 | 0.010 |
| PC value | 0.0010 | all | all | 0.0001 | 0.0010 | all | 0.0010 | all |
| time | 8.84 | 5.09 | 1.40 | 2.92 | 1.63 | 0.84 | 4.94 | 3.58 |
| compl | $50 \%$ | $0 \%$ | $100 \%$ | $50 \%$ | $100 \%$ | $0 \%$ | $100 \%$ | $0 \%$ |
| corr | $33 \%$ | $0 \%$ | $40 \%$ | $11 \%$ | $25 \%$ | $0 \%$ | $6 \%$ | $0 \%$ |

Table 4.8: Summary of the best and worst results obtained from the tests on the set described in 3.6.

Edge, Splines, First method


Figure 4.7: In figures 4.7 a 4.7 b 4.7 c completeness, correctness and computational time for splines analysis for the sets of parameters tested. The vertical lines divide different values of the first parameter. Figure 4.7 d the RMSE is shown for each patch analyzed. The legend displays the number of control point used.

| ctrl pt | 3 x 3 |  | $5 \times 5$ |  | $10 \times 10$ |  | $20 \times 20$ |  |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | Best | Worst | Best | Worst | Best | Worst | Best | Worst |
| diff filter | 0.005 | 0.003 | 0.003 | 0.005 | 0.005 | 0.003 | 0.003 | 0.010 |
| PC value | 0.0010 | 0.0010 | 0.0010 | 0.0001 | 0.0010 | 0.0001 | 0.0010 | all |
| time | 15.25 | 15.42 | 14.65 | 12.88 | 12.13 | 13.42 | 18.59 | 18.65 |
| compl | $100 \%$ | $0 \%$ | $100 \%$ | $50 \%$ | $100 \%$ | $50 \%$ | $50 \%$ | $0 \%$ |
| corr | $25 \%$ | $0 \%$ | $33 \%$ | $5 \%$ | $16 \%$ | $5 \%$ | $7 \%$ | $0 \%$ |

Table 4.9: Summary of the best and worst results obtained from the tests on the set described in 3.6.

In 4.6 d and 4.7 d is visible another increase in the RMSE compared to the values obtained on the edges. Both figures show high peaks for three meshes, two of them maintained at the increase of the approximation flexibility. The spline error is especially irregular for $3 \times 3$ and $4 \times 4$ splines, and generally higher.

The completeness graphs 4.6 a and 4.7 a show how $3 \times 3$ polynomials and $20 \times 20$ splines are not capable to detect all the damages while the other approximants succeed for at least one value of the first parameter. The generally low values of the correctness in figures 4.6 b 4.7 b imply a high number of false positives. In particular the oscillations of the correctness should be noticed for splines in each interval.

The time graphs 4.64 .7 c display a cost that is not half of the border one, even with half the number of patches. In addition the values decrease noticeably for all the approximants with the increase of the first parameter. The $3 \times 3$ polynomial in particular has a cost comparable to the 20x20 one.

The highest peaks in the error graphs corresponds to the cells in the partitioning grid containing the extra curve present in the topmost part of the blades. The partitioning method here does not work correctly, and the patches extracted are too large and irregular. The corresponding edge patch is not simple enough to be modeled with the graph of a function as proven by the fact that increasing the number of variables of the approximating functions does not lower the error in any meaningful way.

In addition to that some portions of those same edges presented a lowered smoothness and missing pieces from the measuring process, increasing the difficulty of building an approximation even more. As a result, a high number of false positives are marked on those zones, lowering the correctness of the results as a whole.

The damages to be found however are on the lower parts of the edges, on patch 16 , the analysis performed still hold values even with the biased correctness parameter.

The result for polynomials are similar to what has been seen previously, with $5 \times 5$ being the safest option for the degrees while 10 x 10 are almost as good but losing in terms of completeness. The lowest degrees are not enough to properly model the surface as we the error and low completeness demonstrate. The $20 \times 20$ polynomials are too flexible and almost fit the damages, detecting them only with the stricter parameters. The computational times also follow what was previously observed, is even more evident the effect of a bad approximation on the cost for the $3 \times 3$ polynomials but other values resent from it too.

Splines instead show comparable completeness values for $10 \times 10$ and $5 \times 5$ control points settings, with also similar amount of correctness and computational times. Splines with $3 \times 3$ control points are not good enough to fit the curved surface and are not reliable in their detection while $20 \times 20$ partially fit the damages too and can locate them only with the stricter first parameter.

On the edges is also more evident the effect of the second parameter involved in the analysis, through the oscillation in the correctness. The reason for it is the shape of the surface on the edge of the mesh.

If the set of points analyzed is on a twisted surface its curvature is affected by the curvature of the surface itself. This means that even undamaged parts present a depth that is not related to an indent and is instead coming from the surrounding surface. Consequently performing the PCA of such point would result in a biased third component.

The effect of the curvature of the surface on the PCA is reduced by using the differences between artificial model and data as third components instead of the points directly. This operation lowers the impact of the surface bends but a curved portion is still more sensitive to small variations than in the "flatter" parts.

A higher value of this parameter is hence necessary on the edges to reduce the number of false positives.

Border approximation error, Second method

| Polynomials | $3 \times 3$ | $5 \times 5$ | $10 \times 10$ | $20 \times 20$ |
| :--- | :---: | :---: | :---: | :---: |
| res | 0.6266 | 0.2278 | 0.0657 | 0.0263 |
| RMSE | $1.2923 \mathrm{e}-5$ | $4.6999 \mathrm{e}-6$ | $1.3565 \mathrm{e}-6$ | $5.4407 \mathrm{e}-7$ |
| ctrl pts | $3 \times 3$ | $5 \times 5$ | $10 \times 10$ | $20 \times 20$ |
| res | 2.1134 | 0.4643 | 0.2095 | 0.1191 |
| RMSE | $7.1450 \mathrm{e}-6$ | $1.5697 \mathrm{e}-6$ | $7.0853 \mathrm{e}-7$ | $4.0282 \mathrm{e}-7$ |

Table 4.10: Residual error and root mean square error for polynomial and splines approximating functions.

The results with the first partitioning method are satisfactory when low degrees polynomials and splines with a relative low number of control points are used. Polynomials are around ten times faster than splines while maintaining the same level of approximation error, completeness and correctness.

The method however is sensitive to high curvature orthogonal to the edges. On those parts the efficacy of the analysis is strongly reduced.

### 4.2.2 Second partitioning method

The second partitioning method for the edges attempt to separate the meshes into pieces with a structure simple enough to be easily approximated with polynomials or splines.

To do that the starting portion of the surface is divided into 4 parts: the two sides that form what we call border and two others that cover the actual edge.

Those portions are built starting from an initial set of mesh points, starting from the same one in the two borders obtained from the first method. From those points a polynomial approximation is built and the mesh points with a small enough approximation error are identified as new starting set. The process is repeated 10 times or until the building set start shrinking.

The resulting border patches are generally smaller because the part where the surface starts bending is left on the edges. The same process is applied to the remaining portion to obtain more manageable portions, step necessary to compensate the parallel strips on the sides of the edge.

This process takes about 6 seconds on one blisk blade, so is about two times slower than the first method.

Once again borders and edges will be studied separately in order to better compare them to the results from the first method.

Border In tables 4.11 and 4.12 and in figures 4.8 and 4.9 are displayed the results of the algorithm on the borders parts obtained with the second method.

Looking at figures 4.8 d do4.9d and comparing them to 4.4 d 4.5 d the RMSE values appear similar for both methods. The tables show how the residual errors are also similar.

The results in term of completeness do not differ much except that the best threshold for the differences are lower and improves the results with 10x10 approximants. An interesting anomaly regard the computational time is the $5 \times 5$ polynomial that for the first time decrease significantly changing the first parameter from 0.03 to 0.05 . In addition to this, it is not capable to detect all the damages without extra findings added to the mix.

The completeness graphs 4.8 b 4.9 b show more elements and higher values compared to the first method, in particular there are optimal results for 10 x 10 polynomials absent for the first method.

Border, Polynomials, Second method


Figure 4.8: In figures 4.8 a 4.8 b 4.8 c completeness, correctness and computational time for polynomial analysis for the sets of parameters tested. The vertical lines divide the graphs in intervals [a,b) based on different values of the first parameter. Figure 4.8d the RMSE is shown for each patch analyzed. The legend displays the degrees of the polynomials used.

| degrees | $3 \times 3$ |  | $5 \times 5$ |  | $10 \times 10$ |  | $20 \times 20$ |  |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | Best | Worst | Best | Worst | Best | Worst | Best | Worst |
| diff filter | 0.005 | 0.003 | 0.005 | 0.020 | 0.005 | 0.010 | all | all |
| PC value | 0.0005 | 0.0001 | 0.0005 | 0.0001 | 0.0001 | 0.0001 | all | all |
| time | 6.56 | 7.98 | 1.16 | 0.30 | 1.30 | 1.14 | 6.38 | 6.57 |
| compl | $100 \%$ | $0 \%$ | $100 \%$ | $0 \%$ | $100 \%$ | $0 \%$ | $0 \%$ | $0 \%$ |
| corr | $28 \%$ | $0 \%$ | $40 \%$ | Nan | $100 \%$ | Nan | Nan | Nan |

Table 4.11: Summary of the best and worst results obtained from the tests on the set described in 3.6.

Border, Spline, Second method


Figure 4.9: In figures 4.9a 4.9b 4.9c completeness, correctness and computational time for splines analysis for the sets of parameters tested. The vertical lines divide different values of the first parameter. Figure 4.9d the RMSE is shown for each patch analyzed. The legend displays the number of control point used.

| ctrl pt | $3 \times 3$ |  | $5 \times 5$ |  | $10 \times 10$ |  | $20 \times 20$ |  |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | Best | Worst | Best | Worst | Best | Worst | Best | Worst |
| diff filter | 0.003 | 0.003 | 0.005 | 0.020 | 0.003 | 0.005 | all | all |
| PC value | 0.0005 | 0.0010 | 0.0010 | 0.0002 | 0.0002 | 0.0001 | all | all |
| time | 26.40 | 26.38 | 26.64 | 16.87 | 20.89 | 19.44 | 35.82 | 36.57 |
| compl | $50 \%$ | $0 \%$ | $100 \%$ | $0 \%$ | $100 \%$ | $0 \%$ | $0 \%$ | $0 \%$ |
| corr | $33 \%$ | $0 \%$ | $40 \%$ | Nan | $40 \%$ | Nan | Nan $\%$ | Nan |

Table 4.12: Summary of the best and worst results obtained from the tests on the set described in 3.6.

Edge approximation error, second method

| Polynomials | $3 \times 3$ | $5 \times 5$ | $10 \times 10$ | $20 \times 20$ |
| :--- | :---: | :---: | :---: | :---: |
| res | 0.7843 | 0.6096 | 0.4762 | 1.0710 |
| RMSE | $4.1525 \mathrm{e}-5$ | $3.2277 \mathrm{e}-5$ | $2.5212 \mathrm{e}-5$ | $5.6704 \mathrm{e}-5$ |
| ctrl pts | $3 \times 3$ | $5 \times 5$ | $10 \times 10$ | $20 \times 20$ |
| res | 1.0696 | 0.6847 | 0.5977 | 0.4984 |
| RMSE | $5.6633 \mathrm{e}-5$ | $3.6251 \mathrm{e}-5$ | $3.1646 \mathrm{e}-5$ | $2.6389 \mathrm{e}-5$ |

Table 4.13: Residual error and root mean square error for polynomial and splines approximating functions.

The computational time for the $3 x 3$ polynomial is even higher than the one for the 20 x 20 polynomials with 0.003 as first threshold. The 5 x 5 approximants also decrease their computational times increasing the first parameter.

Polynomials of degrees $10 \times 10$ detect the damages with low difference filter values and also high levels of correctness earning the position as best choice for polynomial degree, even if only for two parameter settings. The speed is also analog to the $5 \times 5$ polynomials ones, with optimal settings.

Increasing the degrees further causes the fitting of the damages, that consequently can not be found.

The spline results are similar to the polynomial one in the sense that $10 \times 10$ control points show the best results. The splines with $5 \times 5$ control points also get to the same levels of completeness and correctness but with a higher computational time.

The results show how the $3 x 3$ approximants are not flexible enough to fit the sub-meshes. This means that structure of the border patches obtained with the second edge partitioning method are more complex compared to the ones from the previous method.

This is the natural consequence of the methods used to build those patches. While the first method selects based on the flatness, the second one evaluate the proximity to a properly selected polynomial allowing a structure more complex in the patches.

While this fact forces the use of more flexible functions to get optimal results, in the modeling the error committed in the approximation does not increase substantially. This allows a full detection with both splines and polynomials, with only a small increase in the number of false positives.

Edges Tables 4.14 and 4.15 and figures 4.104 .11 display the results obtained on the edge patches extracted with the second partitioning method. Regarding the patches number is worth to remember that the meshes involved are two times the number compared to the first partitioning method ones, because each cell of the grid produce up to two edge pieces instead of one.

Looking at the RMSE in 4.10 d 4.11 d and comparing it to 4.6 d 4.7 d is visible how the number of meshes where the approximation is extremely inaccurate decrease. The portion of the mesh that gives bigger problem is the same for both the methods.

Comparing the tables 4.144 .15 and 4.8 and 4.9 a greater accuracy is shown. The edge patches are then easier to model compared to the one in the first method.

The completeness graph for the polynomials in Fig:4.10a shows how the 10x10 degrees give the best results. Higher and lower degrees detect only half the damages at best, with the exception of $3 x 3$ polynomials that can find all the damages with two set of parameters.

The correctness in Fig:4.10b displays how the combination of settings that detect all the damages also returns a similar amount of false positives. Those erroneous detections are usually

Edge, Polynomials, Second method


Figure 4.10: In figures 4.10a 4.10b 4.10c completeness, correctness and computational time for polynomial analysis for the sets of parameters tested. The vertical line divide the graphs in intervals [a,b) based on different values of the first parameter. Figure 4.10d the RMSE is shown for each patch analyzed. The legend displays the degrees of the polynomials used.

| degrees | $3 \times 3$ |  | $5 \times 5$ |  | 10 x 10 |  | 20 x 20 |  |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | Best | Worst | Best | Worst | Best | Worst | Best | Worst |
| diff filter | 0.003 | 0.005 | 0.010 | 0.020 | 0.005 | 0.010 | 0.003 | 0.020 |
| PC value | 0.0002 | all | all | all | all | all | all | all |
| time | 2.97 | 2.02 | 0.72 | 0.49 | 1.41 | 0.92 | 3.12 | 3.40 |
| compl | $100 \%$ | $0 \%$ | $50 \%$ | $0 \%$ | $100 \%$ | $0 \%$ | $50 \%$ | $0 \%$ |
| corr | $50 \%$ | $0 \%$ | $100 \%$ | $0 \%$ | $40 \%$ | $0 \%$ | $100 \%$ | $0 \%$ |

Table 4.14: Summary of the best and worst results obtained from the tests on the set described in 3.6.

Edge, Spline, Second method


Figure 4.11: In figures 4.11a 4.11b 4.11c completeness, correctness and computational time for splines analysis for the sets of parameters tested. The vertical lines divide different values of the first parameter. Figure 4.11d the RMSE is shown for each patch analyzed. The legend displays the number of control point used.

|  | 3 c |  | 5 x 5 |  | 10 x 10 |  | 20 x 20 |  |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| ctrl pt | 3 x 3 |  | Worst | Best | Worst | Best | Worst |  |
|  | Best | Worst | Best | Worst |  |  |  |  |
| diff filter | 0.005 | 0.003 | 0.003 | 0.020 | 0.010 | 0.020 | 0.003 | 0.005 |
| PC value | 0.0010 | 0.0001 | 0.0005 | all | all | all | 0.0010 | all |
| time | 21.10 | 24.79 | 20.73 | 17.07 | 18.16 | 17.75 | 23.51 | 23.66 |
| compl | $50 \%$ | $0 \%$ | $50 \%$ | $0 \%$ | $50 \%$ | $0 \%$ | $50 \%$ | $0 \%$ |
| corr | $33 \%$ | $0 \%$ | $33 \%$ | $0 \%$ | $16 \%$ | $0 \%$ | $12 \%$ | $0 \%$ |

Table 4.15: Summary of the best and worst results obtained from the tests on the set described in 3.6.
located on the patches where the RMSE is sensibly higher than the average, as for the first method, but they are fewer in number.

The figure 4.10c highlight a sensible decrease in the general analysis time compared to what obtained in the first method (4.6c).

Splines completeness results ?? are inferior to all the previous one, with only half of the damages identified at best. The correctness 4.11 b is also low, meaning a high number of false positives.

The cause for the bad results of $3 \times 5$ and $5 \times 5$ approximants is again the complexity of the surface. The consequent errors cause an increase of the computational time and the number of false positives.

What is especially interesting of these results is the grater differences between polynomial and spline approximants in completeness and correctness.

The main distinction between edge patches for the two methods are their size: the meshes of the second method are roughly half the size of the one from the first. This reduction of data point available reduces the computational time but also increases the relative importance of the damaged zones during the construction of the approximant.

This fact is especially important for the splines functions where each control point influence the function only in a limited portion of the domain. The reason is the B-spline function it is related to, function with limited support by definition.

The second division of the edges allow then a more accurate approximation but reduces the robustness of the splines analysis.

### 4.3 Summary

The results obtained show that the method used is effective in detecting anomalies on the blisk blades. The amount of correct findings depends on the portion analyzed. While the imperfections are always detected the number of false positives varies depending on the accuracy of the approximant and the thresholds used to filter the differences and the normalized third PC value.

For the airfoil and the borders the algorithm shows accurate results with polynomials of degrees $5 \times 5$ or even $3 \times 3$ and the same holds for the number of control points for splines. An extreme low number of false positive and reasonably low computational time makes this algorithm appealing for the application purpose.

Edges are more delicate and require higher degrees and of control points to be analyzed properly. The accuracy is generally lower and a higher number of false positives are detected, concentrated in the part of the blades with a higher amount of twists. A finer partitioning or an increase in the flexibility of the approximant can reduce that, but with the associated risk of overfitting as the 20 x 20 approximants and the splines for the second methods showed.

The first method for edge partitioning, while faster, gives the worst results, and need to be tuned manually to avoid errors. The second one instead is slightly slower, but handles the more complex parts better even if not perfectly. In the next chapter only the second method will be used because of the increase in unpredictability, caused by the corrosion and wearing of the used blades, makes the first method unreliable.

The last important remark to be done is regarding the position of the damages. Since they are located both on the edges and the border patches in some occasion only one of the parts detected the damages. Considering the results of edges and borders together would show higher values for the completeness.

## Chapter 5

## Application of the algorithm to the analysis of worn blades

In this chapter are displayed the results of the algorithm on fourth stage high pressure compressor blades of jet engines. The main difference between this chapter and the previous oneis the level of wear the components were subjected to. Blisks blades were scanned right after the production and before being used to check for manufacturing defect. The scans examined in this chapter were performed on component that had already flew and were discarded at the end of their life-cycle.

In addition to FODs those blades show corrosion and small deformations related to the wearing. The deformations, while small, make the scans too different between each other to use a common reference model for all of them.

The erosion is usually concentrated on the superior part of the blade and the edges, and cause the surface to be coarser. The exterior of the object gains an orange skin look, with both depressed and protruding parts. Those effects reduce the smoothness of the surface and makes the detection of FODs harder, even visually.

The robustness of the method for this kind of additional complications will be tested and the efficacy of the parameters chosen to select the relevant damages will be discussed.

The construction of the grid used for the first part of the partitioning present the main difference from the point of view of the algorithm application. While structure and position of each blisk blade was exactly the same, used blades require an initial control to check if the partitioning grid is correctly positioned.

The base is in the same position for all of them but the superior part might be slightly bent and the edges might fall out of their proper grid's cell. Before applying the algorithm is therefore necessary to visually check the positioning of the blade and eventually adjust the corner points of the grid.

To better compare the results the different parts of the surface are also shown separately. This will allow a comparison with the not-corroded case and between the distinct portions of the blades. Both polynomials and splines functions will be used to perform the approximation. The sets of parameters for the approximating functions, the thresholds for the algorithm and the way the result are shown are the same used previously as described in 3.6.

While not fundamental for the algorithm, it is useful to know the difference in dimension between the blades studied in this chapter and the one in the previous. The scans analyzed in this chapter come from bigger blades and the total number of mesh points is about six times higher than in blisk's case. The number of patches involved is roughly double than for the blisks, and each patch is about twice the width and height. An increase in the residual error and computational time is then to be expected.


Figure 5.1: On the left (5.1a) the 3D model of a fourth stage engine compressor blade, on the upper right (5.1b) a picture of different blades models. On the lower right (5.1c) the superior part of the blade, impact and corrosion are visible.

Airfoil approximation error

| Polynomials | $3 \times 3$ | $5 \times 5$ | $10 \times 10$ | $20 \times 20$ |
| :--- | :---: | :---: | :---: | :---: |
| res | 0.8021 | 0.6304 | 0.4953 | 321.95 |
| RMSE | $6.1381 \mathrm{e}-7$ | $4.8243 \mathrm{e}-7$ | $3.7906 \mathrm{e}-7$ | $2.4636 \mathrm{e}-4$ |
| ctrl pts | $3 \times 3$ | $5 \times 5$ | $10 \times 10$ | $20 \times 20$ |
| res | 1.4346 | 0.6843 | 0.5209 | 0.3035 |
| RMSE | $1.0978 \mathrm{e}-6$ | $5.2368 \mathrm{e}-7$ | $3.9866 \mathrm{e}-7$ | $2.3232 \mathrm{e}-7$ |

Table 5.1: Residual error and root mean square error for polynomial and splines approximating functions.

### 5.1 Airfoil

As for the blisk's case, the airfoil is the easiest part to partition and orientate. Lacking the extra curve on the higher portion present in blisk's blades its shape is even more regular. The curvature changes smoothly and regularly from one edge to the other. The surface is essentially straight in the vertical direction.

The resulting patches are also simple in structure, resembling a saddle point with very slow growth. The approximation is expected to require low degrees of freedom to fit it.

Comparing tables 4.1 and 5.1 show that the RMSE is lower on the compressor blades compared to the blisk's one. Even the residual error is lower, despite the higher number of points involved.

This is true except for $20 \times 20$ polynomials whose error is greater of several order of magnitude. For this reason its RMSE results are not shown in Fig:5.2d, their presence would cause visualization problems for the values of other polynomials.

The computational time in Fig: 5.2c 5.3c lowers with the increase of the first parameter, except for $10 \times 10$ and $5 \times 5$ splines where is almost constant. The time cost is roughly doubled between 3 x 3 and 5 x 5 approximants, and tripled again from 5 x 5 to 10 x 10 . The main differences with the blisk's case is that the values for the 3 x 3 approximant are always lower than the 5 x 5 ones and that the values are generally higher.

The time graphs do not show the values for the $20 \times 20$ approximants because of the difference in magnitude would result poorly graphically. Logarithmic axis ratios would have shown their values acceptably but also made the comparison with others graphs harder and was then avoided. The computational time necessary for those polynomials was 250 seconds while splines required 1500 seconds.

Figures 5.2 d and 5.3 d show how the RMSE is higher on the last patches, the one on the superior part, where the corrosion is higher. The increase of the flexibility of the approximant reduces the error, but the gain is quite limited except from $3 x 3$ to $5 x 5$ splines. A periodic oscillation in the error values is also visible for groups of roughly 20 patches.

The completeness graphs in Fig: 5.2a and 5.3a show lower values overall if compared to 4.2 a and 4.3 a , especially for higher values of the first parameter. The correctness values in 5.2 b 5.3 b are also lower than the blisk's ones.

Increasing the second parameter reduce the number of findings but increase the correctness. This effect is more evident here than it was for the blisk's case.

The reason behind the lower approximation error is mainly the simpler geometric structure of the surface, due to the lack of extra curves. The presence of corrosion increases the error in the patches affected but the change is not high enough to spoil further analysis.

The periodic oscillation in the error depends on the position of the patch. Because of the curve of the blade, and the way the grid divides it, the patches on one side are bigger to the one

## Airfoil, Polynomials



Figure 5.2: In figures 5.2a 5.2b 5.2c completeness, correctness and computational time for polynomial analysis for the sets of parameters tested. The vertical lines divide the graphs in intervals [a,b) based on different values of the first parameter. Figure 5.2d the RMSE is shown for each patch analyzed. The legend displays the degrees of the polynomials used.

| degrees | $3 \times 3$ |  | $5 \times 5$ |  | 10 x 10 |  | 20 x 20 |  |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | Best | Worst | Best | Worst | Best | Worst | Best | Worst |
| diff filter | 0.005 | 0.020 | 0.003 | 0.020 | 0.003 | 0.020 | 0.003 | 0.020 |
| PC value | 0.0002 | all | 0.0002 | all | 0.0002 | all | 0.0002 | 0.0010 |
| time | 8.28 | 2.52 | 9.93 | 6.40 | 29.38 | 26.35 | 270.41 | 280.91 |
| compl | $94 \%$ | $11 \%$ | $100 \%$ | $11 \%$ | $100 \%$ | $5 \%$ | $47 \%$ | $11 \%$ |
| corr | $48 \%$ | $66 \%$ | $43 \%$ | $66 \%$ | $41 \%$ | $100 \%$ | $1 \%$ | $<1 \%$ |

Table 5.2: Summary of the best and worst results obtained from the tests on the set described in 3.6.

## Airfoil Spline



Figure 5.3: In figures 5.3a 5.3b 5.3c completeness, correctness and computational time for splines analysis for the sets of parameters tested. The vertical lines divide different values of the first parameter. Figure 4.5 d the RMSE is shown for each patch analyzed. The legend displays the number of control point used.

| ctrl pt | 3 x 3 |  | $5 \times 5$ |  | 10x10 |  | 20x20 |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | Best | Worst | Best | Worst | Best | Worst | Best | Worst |
| diff filter | 0.003 | 0.020 | 0.003 | 0.020 | 0.003 | 0.020 | 0.003 | 0.020 |
| PC value | 0.0002 | all | 0.0002 | all | 0.0002 | all | 0.0002 | all |
| time | 90.07 | 57.81 | 102.96 | 98.95 | 359.74 | 356.49 | 1521 | 1529 |
| compl | 100\% | 11\% | 100\% | 11\% | 100\% | 5\% | 94\% | 0\% |
| corr | 44\% | 66\% | 45\% | 66\% | 38\% | 100\% | 28\% | Nan |

Table 5.3: Summary of the best and worst results obtained from the tests on the set described in 3.6.
in the other. The bigger patches are slightly harder to model but the error increase does not affect the analysis in a remarkable way.

Effects of the first threshold on the computational time are more apparent here because the higher amount of irregularities caused by corrosion.

Increasing the first parameter filters sizable amounts of points even for high degree polynomials because the number of irregularities (i.e. FODs and corrosion) is higher than the approximant fitting capability. This behavior is exactly the desired one: a good general fit and little modeling of the imperfections.

Splines are more flexible because of their structure and only the $3 x 3$ one shows clear variations in time cost. Combined with the higher RMSE is reasonable to suppose that the number of control points is too low, or the bare minimum necessary, to model the surface.

Increasing the first parameter lower the completeness of all approximants for even small variations. This happens because the size of the damages to be found is small enough to be erroneously filtered out.

The second parameter variations have a more evident effect here compared to the blisks because there are many more imperfections to be discarded based on their shape. Corrosion can still be marked as damage if deep enough and can be distinguished only using the second parameter. Increasing its value is hence an effective way to increase the correctness however less damages are found decreasing the completeness. The damages lost this way are the "flatter" one, the most similar to corrosion.

The behavior the $20 \times 20$ polynomial is explainable with a wrong approximation. The approximating polynomials oscillate to try and fit all the small bumps on the surface resulting in a problem similar to the Runge phenomenon. While about half of the damages are usually identified the number of false positives is more than a hundred times the true positives.

Summarizing, the use of 3 x 3 or 5 x 5 approximants with the lowest parameters gives the best results for completeness and computational time. Those results are generally worse than what obtained on the pristine blisk's blades. Increasing the second threshold trade part of the completeness for a higher correctness.

### 5.2 Edges

The same consideration exposed in 4.2 still stand for blades: dividing correctly airfoil and edges is difficult and corrosion makes the difference even fuzzier. A simple method like the first one is not flexible enough to react to the alteration that might incur, the second method showed more promising results.

The patches of mesh adjacent to the edges were treated separately (as "border") from the airfoil because of the different structure. The junction part between the two sides of the airfoil is referred as edge.

The two lateral edges are inclined differently with respect to the partitioning grid direction and the sub meshes extracted have different shapes. In particular the border patches have different sizes between the two edges and even between the ones in the same grid cell.

The damages on these portions of the blades are usually located on the edge or in their proximity. This makes the border part closer to the edges focal for the analysis, increasing the importance of a correct partitioning.

The damages searched on the blisk's edges were big enough to appear on both edge and border so that both parts allowed their detection. In the compressor blade's case the damages are instead smaller and in general can not be detected in both parts. For this reason the additional table 5.10 will show the combined results obtained from edges and borders, using the sets of parameters that allowed the best results on each.

## Border approximation error, Second method

| Polynomials | $3 \times 3$ | $5 \times 5$ | $10 \times 10$ | $20 \times 20$ |
| :--- | :---: | :---: | :---: | :---: |
| res | 1.1503 | 0.4720 | 0.1812 | 30.3524 |
| RMSE | $7.0399 \mathrm{e}-6$ | $2.8890 \mathrm{e}-6$ | $1.1090 \mathrm{e}-6$ | $5.3449 \mathrm{e}-5$ |
| ctrl pts | 3 x 3 | $5 \times 5$ | 10 x 10 | 20 x 20 |
| res | 1.8648 | 0.8287 | 0.2476 | 0.0952 |
| RMSE | $1.1412 \mathrm{e}-6$ | $5.0717 \mathrm{e}-6$ | $1.5153 \mathrm{e}-6$ | $5.8315 \mathrm{e}-7$ |

Table 5.4: Residual error and root mean square error for polynomial and splines approximating functions.

### 5.2.1 Border

Looking at table 5.4 and comparing it to the one for the airfoil 5.1 is evident an higher error of approximation. Even if the number of patches is about one fifth of the airfoil the residual error is higher and the RMSE is also doubled. Those values are instead comparable with the results for the blisk's borders.

The 20x20 polynomials show an error several orders of magnitude higher than the other polynomials, or splines, like it did for the airfoil. For graphical reasons it is not shown in Fig:5.4d, as explained in the previous section. That same figure and Fig:5.5d shows how increasing the degree or number of control points reduce the RMSE. The spline graph also shows a difference between the first and second half of the values for the $3 \times 3$ spline. The two groups correspond to the border patches of the left and right edges respectively.

The time graphs Fig:5.4c and 5.5 c display an increase of the cost along with the increase of the number of approximant parameters. The $3 \times 3$ and $5 \times 5$ models present a decrease in computational time with the increase of the first parameter.

The completeness graphs 5.4 a and 5.5 a show a drastic increase in the number of missed findings, with polynomials at $\sim 40 \%$ and splines at $\sim 60 \%$. The correctness graphs additionally show how the number of false positives is up to four time the number of correct ones. This is a poor result compared to the airfoil or blisks in general.

The reason for the decrease in computational time with the increase of the first parameter has been explained multiple times, this low completeness results are new. The main reason behind them is the location of those damages. Most of them are in fact in the zones between edges and borders, and are then left on only one of the two parts by the partitioning. So, many damages are then absent or only partially present on the border, as will happen with the edges.

Even if the damage is present on the border pieces its position makes it hard to be detected. The part of the border closest to edge is the most difficult to model correctly because of its small relative size and rapid change in curvature. Damages on this part of the mesh will be harder to distinguish from the approximation error.

In addition their small size implies that an increase of the first parameter can easily filter them out, as shown with the airfoil. The value of the second threshold plays therefore an important role in excluding the false positives.

The use of $5 \times 5$ or $10 \times 10$ splines with low values for the first and second parameters is what appear to be the best strategy to maximize the number of findings. Increasing the second threshold is once again a way to trade completeness for a lower number of false positives.

Border, Polynomials, second method


Figure 5.4: In figures 5.4a 5.4b 5.4c completeness, correctness and computational time for polynomial analysis for the sets of parameters tested. The vertical lines divide the graphs in intervals [a,b) based on different values of the first parameter. Figure 5.4 d the RMSE is shown for each patch analyzed. The legend displays the degrees of the polynomials used.

| degrees | 3 x 3 |  | 5 x 5 |  | 10 x 10 |  | 20 x 20 |  |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | Best | Worst | Best | Worst | Best | Worst | Best | Worst |
| diff filter | 0.005 | 0.020 | 0.005 | 0.020 | 0.003 | 0.020 | 0.003 | 0.010 |
| PC value | 0.0005 | all | 0.0002 | all | 0.0002 | all | 0.0002 | all |
| time | 15.72 | 0.39 | 5.07 | 0.88 | 4.32 | 3.44 | 30.50 | 25.16 |
| compl | $30 \%$ | $0 \%$ | $38 \%$ | $0 \%$ | $38 \%$ | $0 \%$ | $23 \%$ | $0 \%$ |
| corr | $100 \%$ | Nan | $83 \%$ | Na | $50 \%$ | Nan | $1 \%$ | Nan |

Table 5.5: Summary of the best and worst results obtained from the tests on the set described in 3.6.

Border, Spline, Second method


Figure 5.5: In figures 5.5a 5.5b 5.5c completeness, correctness and computational time for splines analysis for the sets of parameters tested. The vertical lines divide different values of the first parameter. Figure 5.5 d the RMSE is shown for each patch analyzed. The legend displays the number of control point used.

| ctrl pt | 3 x 3 |  | $5 \times 5$ |  | 10x10 |  | 20x20 |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | Best | Worst | Best | Worst | Best | Worst | Best | Worst |
| diff filter | 0.005 | 0.020 | 0.003 | 0.020 | 0.003 | 0.020 | 0.003 | 0.020 |
| PC value | 0.0005 | all | 0.0002 | all | 0.0002 | all | 0.0002 | all |
| time | 20.91 | 8.86 | 29.63 | 10.81 | 32.51 | 29.24 | 119.43 | 119.27 |
| compl | 38\% | 7\% | 53\% | $0 \%$ | 46\% | $0 \%$ | 30\% | 0\% |
| corr | 83\% | 100\% | 28\% | Nan | 27\% | Nan | 33\% | Nan |

Table 5.6: Summary of the best and worst results obtained from the tests on the set described in 3.6.

Edge approximation error, Second method

| Polynomials | $3 \times 3$ | $5 \times 5$ | $10 \times 10$ | $20 \times 20$ |
| :--- | :---: | :---: | :---: | :---: |
| res | 1.0428 | 0.6505 | 0.4592 | 0.3399 |
| RMSE | $1.1607 \mathrm{e}-5$ | $7.2409 \mathrm{e}-6$ | $5.1117 \mathrm{e}-6$ | $3.7838 \mathrm{e}-6$ |
| ctrl pts | $3 \times 3$ | $5 \times 5$ | $10 \times 10$ | $20 \times 20$ |
| res | 1.2369 | 0.7484 | 0.6050 | 0.2714 |
| RMSE | $1.3767 \mathrm{e}-5$ | $8.3308 \mathrm{e}-6$ | $6.7347-6$ | $3.0208 \mathrm{e}-6$ |

Table 5.7: Residual error and root mean square error for polynomial and splines approximating functions.

### 5.2.2 Edge

The RMSE values in table 5.7 are about double the border ones and one order of magnitude higher than the airfoil's. The residual error is also much higher if the difference in the number of points involved is considered. The RMSE graphs shows once again how the increase in flexibility of the approximating function reduce the error.

The computational time behave as usual, with the only difference that all the approximants reduce their time cost increasing the first parameter. The cost associated with the 3 x 3 polynomial is especially high, being comparable to the slowest $20 \times 20$ one.

The completeness and correctness graphs display results similar to the borders one except for the number of parameters of the best approximant. The highest percentage of detection is at $38 \%$ for polynomial and $61 \%$ for splines, both obtained with $20 \times 20$ approximants. Correctness is also low compared to the airfoil and the blisk edges, being instead more similar to the border results.

The structure of the edges is harder to model correctly and require more flexibility from the parameterization as the results for the 20 x 20 functions show. The patches used for blades are roughly double the size of the blisk's ones and this helps in keeping the approximant smoother, but require more flexibility compared to the previous chapter case. The 3 x 3 and 5 x 5 approximants are then clearly insufficient and this is shown by the fact that even when the high completeness is achieved the correctness is abysmal.

Splines with 20x20 control points with low values of the two thresholds seems to be the best solution to detect as many damages as possible, even if the computational time cost is relatively high.

Looking now at table 5.10 it can be see that even when merging the results for borders and edges, and choosing the best parameters, not all the damages are correctly detected. By using different numbers of parameters for the approximants (best column) the polynomials show the best results at $69 \%$ completeness and $52 \%$ correctness, splines obtain slightly lower results.

The increase in completeness for the combined findings demonstrate how some damages were detected only by one of the between edges and borders. A more accurate control of the missed damages showed how the meshes they are in are divided incorrectly by the partitioning algorithm. The returned meshes where fragmented, and made the constructed approximants resulted inaccurate, highlighting the importance a correct partitioning process.

### 5.3 Top part

The upper row of the partitioning grid covers the topmost part of the blade where the two sides of the airfoil get connected by a thin horizontal strip, orthogonal to both of them.

Edge, Polynomials, Second method


Figure 5.6: In figures 5.6a 5.6b 5.6c completeness, correctness and computational time for polynomial analysis for the sets of parameters tested. The vertical lines divide the graphs in intervals $[\mathrm{a}, \mathrm{b})$ based on different values of the first parameter. Figure 5.6d the RMSE is shown for each patch analyzed. The legend displays the degrees of the polynomials used.

| degrees | 3 x 3 |  | 5 x 5 |  | $10 \times 10$ |  | 20 x 20 |  |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | Best | Worst | Best | Worst | Best | Worst | Best | Worst |
| diff filter | 0.020 | 0.003 | 0.020 | 0.003 | 0.010 | 0.003 | 0.005 | 0.003 |
| PC value | 0.0005 | 0.0010 | 0.0010 | 0.0010 | 0.0001 | 0.0001 | 0.0002 | 0.0010 |
| time | 1.10 | 18.97 | 0.73 | 14.08 | 2.39 | 9.63 | 12.61 | 19.31 |
| compl | $23 \%$ | $0 \%$ | $15 \%$ | $11 \%$ | $30 \%$ | $7 \%$ | $38 \%$ | $7 \%$ |
| corr | $100 \%$ | $0 \%$ | $100 \%$ | $66 \%$ | $80 \%$ | $6 \%$ | $45 \%$ | $11 \%$ |

Table 5.8: Summary of the best and worst results obtained from the tests on the set described in 3.6.

Edge, Spline, Second method


Figure 5.7: In figures 5.7a 5.7b 5.7c completeness, correctness and computational time for splines analysis for the sets of parameters tested. The vertical lines divide different values of the first parameter. Figure 5.7d the RMSE is shown for each patch analyzed. The legend displays the number of control point used.

| ctrl pt | 3 x 3 |  | 5 x 5 |  | 10 x 10 |  | 20 x 20 |  |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | Best | Worst | Best | Worst | Best | Worst | Best | Worst |
| diff filter | 0.003 | 0.020 | 0.003 | 0.020 | 0.003 | 0.020 | 0.003 | 0.020 |
| PC value | 0.0002 | 0.0001 | 0.0002 | all | 0.0002 | all | 0.0002 | all |
| time | 28.27 | 2.52 | 25.28 | 5.81 | 23.52 | 11.51 | 47.21 | 44.25 |
| compl | $61 \%$ | $15 \%$ | $53 \%$ | $7 \%$ | $53 \%$ | $7 \%$ | $61 \%$ | $0 \%$ |
| corr | $8 \%$ | $13 \%$ | $6 \%$ | $33 \%$ | $8 \%$ | $100 \%$ | $29 \%$ | Nan |

Table 5.9: Summary of the best and worst results obtained from the tests on the set described in 3.6.

Border and edge combined results

| Polynomials | $3 \times 3$ | $5 \times 5$ | $10 \times 10$ | $20 \times 20$ | best |
| :--- | :---: | :---: | :---: | :---: | :---: |
| Completeness | 53 | 46 | 61 | 61 | 69 |
| Correctness | 100 | 75 | 53 | 3 | 52 |
| ctrl pts | $3 \times 3$ | $5 \times 5$ | $10 \times 10$ | $20 \times 20$ | best |
| Completeness | 38 | 38 | 46 | 53 | 61 |
| Correctness | 33 | 20 | 27 | 2 | 27 |

Table 5.10: Completeness and correctness with borders and edges result combined. Polynomials and splines results with the best setting for each part and fixed number of approximation parameters. The last column shows the best results with polynomials and splines using different degrees/number of control points.

This thin portion of the mesh is not analyzed because of low interest for the application. What is instead important to analyze is the airfoil part close to this edge and the junction part between sides and horizontal strip. Being both on the upper part and close to an edge the top part is subject to high amount of corrosion and deformations, especially near the corners.

The same considerations made for the edges are still valid for the top too, with the exception that the actual edge is here discarded. What is being analyzed are the equivalent of borders for the edges, and the results show this similarity.

Table 5.11 shows how the RMSE is comparable to the one obtained on the borders, and Fig:5.8d 5.9d further confirm it. The residual values are also approximately proportional to the number of patches. As for airfoil and borders the $20 \times 20$ polynomial has error values too high to be properly displayed.

The time graphs Fig:5.8c and 5.9 c show the same structure as the ones for the borders with slightly lower values and decreasing costs for the 10 x10 approximant.

The completeness graphs show the first important difference. While borders detected at best half of the relevant damages, on the top portion all the damages can be found with the right set of parameters. On the other hand the correctness values for those same parameters are lower than twenty, meaning a high number of false positives.

As for the borders $5 \times 5$ and $10 \times 10$ approximants gives the best results, with similar computational costs.

Those results are caused by the same problems described for the borders. The difference in completeness depends on the position of the damages. The FODs are not concentrated in proximity of the edges but spread more uniformly on the patches.

The critical part close to the top horizontal strip is still approximated less accurately than the remaining portion, but fewer damages are there, making it less critical. The higher complexity of the patches shapes requires higher flexibility from the model to fit correctly and the presence of corrosion decrease the correctness.

Here too the choice of the second parameter determine the ratio between number of findings and their correctness. The high number of false positives is due to the high amount of corrosion damages present in this part, only a correct selection of the second threshold is able to filter most of them.

### 5.4 Summary

The results on worn blades show how the corrosion and deformations reduce the overall accuracy of the algorithm compared to the ones on newly manufactured blisks. This was to

Top part, Polynomials


Figure 5.8: In figures 5.8a 5.8b 5.8c completeness, correctness and computational time for polynomial analysis for the sets of parameters tested. The vertical lines divide the graphs in intervals $[\mathrm{a}, \mathrm{b})$ based on different values of the first parameter. Figure 5.8d the RMSE is shown for each patch analyzed. The legend displays the degrees of the polynomials used.

| degrees | $3 \times 3$ |  | $5 \times 5$ |  | $10 \times 10$ |  | $20 \times 20$ |  |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | Best | Worst | Best | Worst | Best | Worst | Best | Worst |
| diff filter | 0.003 | 0.020 | 0.003 | 0.020 | 0.005 | 0.020 | 0.010 | 0.003 |
| PC value | 0.0002 | all | 0.0005 | all | 0.0002 | all | 0.0010 | all |
| time | 10.27 | 0.66 | 7.92 | 0.90 | 3.27 | 2.15 | 19.36 | 19.60 |
| compl | $100 \%$ | $33 \%$ | $100 \%$ | $33 \%$ | $100 \%$ | $0 \%$ | $33 \%$ | $0 \%$ |
| corr | $10 \%$ | $33 \%$ | $60 \%$ | $100 \%$ | $15 \%$ | Nan | $1 \%$ | Nan |

Table 5.12: Summary of the best and worst results obtained from the tests on the set described in 3.6.

Top part, Spline


Figure 5.9: In figures 5.9a 5.9b 5.9c completeness, correctness and computational time for splines analysis for the sets of parameters tested. The vertical lines divide different values of the first parameter. Figure 5.9 d the RMSE is shown for each patch analyzed. The legend displays the number of control point used.

| ctrl pt | 3 x 3 |  | 5x5 |  | 10x10 |  | 20x20 |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | Best | Worst | Best | Worst | Best | Worst | Best | Worst |
| diff filter | 0.003 | 0.020 | 0.005 | 0.020 | 0.003 | 0.020 | 0.005 | 0.020 |
| PC value | 0.0002 | all | 0.0002 | all | 0.0005 | all | 0.0002 | all |
| time | 12.31 | 5.24 | 13.64 | 8.82 | 34.28 | 28.67 | 121.57 | 120.29 |
| compl | 66\% | 0\% | 100\% | $33 \%$ | 100\% | 25\% | 100\% | 0\% |
| corr | 16\% | Nan | 10\% | 100\% | 33\% | 100\% | 20\% | Nan |

Table 5.13: Summary of the best and worst results obtained from the tests on the set described in 3.6.

Top part approximation error

| Polynomials | $3 \times 3$ | $5 \times 5$ | $10 \times 10$ | $20 \times 20$ |
| :--- | :---: | :---: | :---: | :---: |
| res | 0.6411 | 0.4644 | 0.3522 | 62.5028 |
| RMSE | $6.7676 \mathrm{e}-6$ | $4.9023 \mathrm{e}-6$ | $3.7174 \mathrm{e}-6$ | $6.5971 \mathrm{e}-4$ |
| ctrl pts | $3 \times 3$ | $5 \times 5$ | $10 \times 10$ | $20 \times 20$ |
| res | 1.0795 | 0.5428 | 0.3913 | 0.2633 |
| RMSE | $1.1394 \mathrm{e}-5$ | $5.7295 \mathrm{e}-6$ | $4.1302 \mathrm{e}-6$ | $2.7799 \mathrm{e}-6$ |

Table 5.11: Residual error and root mean square error for polynomial and splines approximating functions.
be expected given how the presence of corrosion makes the detection of the FODs harder even visually, performed by human operators.

In particular on edges, borders and the topmost portion the method has difficulties in detecting all the damages, especially the one located between edges and borders. Increasing the degree of freedom of the approximant and using the more flexible splines proves to be the correct approach to maximize the findings.

The airfoil part instead allow all the damages to be detected even with $3 \times 3$ and $5 \times 5$ degree polynomials, making the process computationally cheap. All parts show how choosing the correct values for the parameters is fundamental.

By changing the first parameter, even slightly, a substantial number of imperfections are added or removed from the findings. This happens because the corrosion and the impacts often have similar small depths, making the filtering threshold a delicate parameter.

Even more sensitive is the second threshold, the minimum value for the normalized third PC value. The completeness and correctness graphs show how increases in this parameter are the simplest way to filter out most of the false positives, at the cost of a lower number of true positives. Keeping it low allow for more FODs to be detected, along with the corroded parts with a more impact-like appearance.

One last remark deals with the patches dimension. Compared to the blisks, the sub-meshes analyzed in the fifth chapter contained four times the number of points and area. This difference seems to not affect the algorithm but is actually important. Tests performed on smaller patches on engine blades resulted in poorer results because the distorting effect of corrosion on the surface structure was not mediated by a higher amount of data. On blisks this was not visible because of the lack of corrosion.

For similar reasons smaller patches increase the risk of overfitting the damages because the points on the impacts increase in relative importance. Using bigger patches then helps in building a smoother approximant but might require more flexible models. This "hidden" parameter must be decided based on the surface complexity and the level of corrosion.

## Chapter 6

## Conclusions and improvement ideas

In this work the issue of foreign object damages detection on jet engine blades has been addressed. The desired procedure should be as automatized as possible and adaptable to the shapes of different blade models.

As already stated impact damages present themselves in a variety of shapes and dimensions, which makes it difficult to define general criteria for their identification. The corrosion affecting the worn objects reduces the smoothness of their surface and the distinction between impact and erosion damages is required. Once again this difference is hard to define quantitatively, and even visually the distinction is not always clear.

The deformations caused by mechanical stress, impacts and use in general are bigger than the minimum size of the searched damages. Adjustments on the scan to fit it to a reference model are difficult for the lack of reliable reference points on it, making the use of models impractical.

By looking at the results obtained it is possible to identify the improvable parts of the method and the features that instead guarantee correct results.

### 6.1 Summary

The algorithm proposed works with small portions of the surface at a time, building a smooth approximation and using it as an artificial reference model to detect the irregularities. The potential impact locations are then analyzed using the PCA to determine their shapes and classified to distinguish between FODs and corrosion.

The local nature of the method allows the application of it to different blades models, requiring only the construction of a proper initial partitioning grid to divide the different components, i.e. airfoil, edges, top and fillet. ${ }^{1}$

The method proposed has been tested on newly manufactured blisk blades to detect production defects and on worn compressor blades to detect FOD.

### 6.1.1 Blisk summary

In the blisk case there was no corrosion and the use-induced deformations were absent, reducing the overall complexity of the task to the only detection of imperfections. This analysis highlighted the detection part of the algorithm, showing its performance in the identification of generic imperfections.

[^4]The analysis on worn compressor blades instead shows the results for the intended application. Corrosion and deformations are both present and affect the scan, reducing the FODs visibility and requiring the classification of the damages between impacts and erosion.

The main variables of the analysis consist in the choice of the approximating function, the minimum threshold value for the difference between data and approximation (first parameter) and the minimum value for the third component of the normalized principal components values (second parameter).

The choice of the approximant flexibility is dependent on the patch structure analyzed, requiring more flexible functions for meshes with a more complex geometry. The approximant should be the one with the minimum degrees of freedom necessary to model the structure of the patch, increasing the degrees reduces the approximation error but increase the risk of fitting the damages spoiling the analysis.

The first parameter depends on the minimum depth of the damages to be found and the error level of the approximation. The smaller the damage the lower the parameter, the higher the mean difference the higher the parameter to filter out the error.

The second parameter determines the shape of the damages identified as FODs. The higher its value, the more spherical looking are the damages marked as impacts.

By using the correct sets of approximating functions and filtering parameters all the damages where detected on the blisks and the majority was identified on engine blades.

The results on the blisks shows how, in absence of corrosion, the detection procedure is reliable. This holds especially on the airfoil, where all the damages are found for a wide range of filtering parameters and approximants.

The structure of borders and edges is more complex than the almost flat airfoil and requires higher degrees of freedom for the approximant. The parameter choice is also a more sensitive decision, as shown by the vastly different values obtained for completeness and correctness. The number of false positives is generally higher in those portions, especially on the edges.

The result's quality is mainly dependent from the approximating function and first parameter. In particular low degrees polynomials and low values for the first parameter work for borders and airfoil, while the edge requires higher degrees and values to model it correctly. The effect of the second parameter are minor because there is no need to distinguish the type of damages involved, every imperfection big enough have to be located.

### 6.1.2 Worn blades summary

The results of chapter five show how most of the damages are detected on worn out blades, even if not as fully as for the blisks. The number of false positives is also generally higher as some of the corrosion damages are extremely similar to the impact ones. Looking at the different parts of the blade similar behaviors are shown: on the airfoil the algorithm accuracy is higher, followed by borders and top, on edges is the less precise.

The airfoil shows the best results in terms of FODs identification, pointing out that the same type of approximant used for blisk airfoils works for worn blades. The presence of corrosion increase the number of false positive detected, as some of the deeper and rounder corroded parts are identified as impact damages.

The same happens on the top part, with a higher relative number of false positives.
Edges and borders are the real problematic portions. Here the completeness and correctness values are lower compared to the blisk counterpart, mainly because the smaller damages are not identified on both parts. In fact the combined findings shown in table 5.10 display an increase in the quality of the findings. As for the blisks the choice of the parameters can change substantially the number of correct findings and false positives.

The approximating function and first parameter choice follow the same general rules stated before, with additional attention on the size of the minimal damage to be found and the level of corrosion. As for the blisk, on the edges more flexible approximants and higher first parameter values are needed.

On the used blades the effect of the second filtering parameter is more evident and such sensitivity is caused by the presence of corrosion. The two kind of damages in fact often show similar depth values, and can not then be easily filtered out with the first threshold, while the shape of the damages is instead determined using the PCA. By adjusting the second parameter value the correctness and completeness of the algorithm can be traded.

To detect more damages the second parameter must be keep low, but this way the number of corrosion damages identified as FODs also increase. Increasing the parameter reduces slightly the number of impacts detected, the more shallow and "corrosion-like" ones, but also reduce substantially the number of false positives (e.g see Fig: 5.3a 5.3a on the first column).

### 6.2 Outlook

The algorithm proposed, while giving good results on most of the surface, shows drops in accuracy near specific parts, namely the edges. This behavior could be corrected perfecting some parts of the method, mainly improving the partitioning and approximant construction procedures.

Regarding this, some ideas are here proposed, based on the observation of the algorithm results. The outlined directions for further research are the most likely to improve the method without altering its basic structure.

### 6.2.1 Partitioning method

Airfoil and top parts exhibit accurate results, while the edges (and borders) are often too inaccurate. The main problem with the edges is the partitioning: the structure of the extracted patches is sometimes too irregular.

Finding a better partitioning method is certainly one of the improvable aspects of the method. As the test results showed, the quality of the results improved drastically on those parts of the edge where the extracted sub-meshes were more regular. Where the partitioning returned fragmented, or oddly shaped meshes, it was more likely to miss damages and find more false positives.

A good partitioning method should produce patches regular in size, shape and easily fitted by the approximants. At the same time the portions should be kept wide enough to allow the approximant to ignore the minor damages. The method should also be flexible enough to work even with deformations and for different shapes of the edges portions.

The second edge partitioning method is a step in the right direction but can certainly get better. The iterative procedure used offer a good degree of flexibility and reduce the sensitivity to reference direction compared to the first one. However that same iterative nature makes the process unpredictable, sometimes disconnected portions or oddly shaped patches were extracted.

The addition of some constraints to the updated building set could help in solving the problem, the experiments done did not produce substantial improvements without increasing excessively the time needed. Further research is then required.

Another step that could be improved is the grid used for the partitioning. The method used for the grid construction is extremely simple in order to be more easily adapted to different blade models but could be improved with relative simplicity. As an example, the extra curve on the


Figure 6.1: On the left an artificial model of a surface with an impact in the center and some slight corrosion. In the middle the (absolute value of) differences between data and a spline approximant. On the right the differencies with a penalized spline approximant.
top of blisk blades makes that part more complex, a finer partitioning with a tighter grid could help in simplifying its structure.

### 6.2.2 Approximating function construction

The construction of the approximant and its choice is one part of the method that could be refined. For the test results presented in the previous chapters the least square method was used to determine the approximant parameters minimizing 2.4 or 2.6 . This method, while fast and generally effective, is not optimal.

The problem comes from the fact that the damaged parts are still considered in the cost functions and influence the final result, causing the approximant to at least partially model them. This makes, as an example, the differences between data and model inaccurate to determine the depth of the damage, and reduces the accuracy of the PCA to determine the shape.

To solve this problem two solutions could be tried: changing the cost function or applying a selection to the data points used.

Adding a penalty weight as explained in 2.3.1 is one possible solution to obtain a smoother result. If the surface approximant is smooth enough the damages can not be fitted and the model is useful for a comparison. The test performed in this direction showed how the damaged zones got smoothed out but the error increased in the portions of the patch with steep inclinations. As shown in figure 6.1 the parts with higher gradient, the borders, are not fitted correctly and the error value is similar to the difference on the damaged part. The penalized approximation loses then most of its value as a reference model.

The tests with different kind of penalty weights or smoothing factors did not improve this behavior, especially on the most sensitive parts, like the edges and borders, where the surface keeps changing inclination.

An additional experiment performed was to increase the weight of parts of the patches based on the normal direction. By giving more importance to the triangles with surface normal that made wider angles with the vertical direction, the critical part of the borders and the sides of the edges were better fitted. The approximation on the remaining part of the surface however worsened sensibly making this modification harmful for the algorithm purpose.

A possible way to select a more meaningful set of data to build the approximation is to perform the analysis iteratively. Removing or reducing the weight of the portions identified as damages the next approximation should skip them completely, allowing for the fitting of only the undamaged parts (or with corrosion noise). The speed of the analysis, especially with


Figure 6.2: Example of the result of the parabola-parametrization. On the left the first method, on the right the more refined one. In blue the initial points(projected on the $y z$ plan) in yellow the rotated one, in red the best fitting parabola and in green the points in the new coordinates.
polynomials, guarantees that such a solution would not be extremely costly in computational terms, but the effects of an initial detection of an incorrect big damage require further research.

A last modification to the approximation definition method could be adapting the number of approximant parameters patchwise by analyzing a scan of an undamaged blade partitioned with the same grid. By analyzing the root mean square error on a pristine scan a more precise idea of the approximant structure could be obtained, and also help in deciding if the size and shape of the grid cells is correct or should be changed.

As of now the distinction is only based on the structure of the portions (airfoil,borders...) and do not consider information on the size of the patch or more specific detail like the curve on the upper part of the blisk blades. This approach was motivated by the desire of keeping the algorithm simple and easily adaptable to different blade models and still showed good results. However it could be interesting to see if such modification would improve the method performance.

## Alternative parameterization

Another possible approach to reduce the problems related to edge portions is to use a different parameterization of their points. A transformation capable of flattening this part of the surface would solve most of the issues related to its approximation. For this reason different parameterizations has been tested on the curved portion of the edge.

In this subsection the patches are supposed already oriented along the PC direction so that the first coordinate moves along the length of the axis. Looking instead at Fig: 6.2 the second coordinates is represented on the horizontal axis, the third on the vertical one.

The first and probably more natural one is the parameterization with cylindrical coordinates

$$
\begin{equation*}
(x, y, z) \longrightarrow\left(x, \arctan \left(\frac{z}{y}\right), \sqrt{y^{2}+z^{2}}\right)=(x, \theta, \rho) \tag{6.1}
\end{equation*}
$$

which allowed a better approximation on the curved part of the edge but it worsened sensibly closer to the borders. On those parts the points are in fact mapped in extremely steep surfaces, nullifying any vantage obtained

A similar kind of re-parametrization for the edges was based on the idea of mapping the best approximating parabola ${ }^{2}$ into an horizontal line, and moving along the nearby points.

[^5]The transformation for the parabola's points uses the length $L(y)$ of the curve representing it $\gamma(y)=\left(y, a y^{2}+b y+c\right)$ i.e.

$$
L(y)=\int_{0}^{y}\|\dot{\gamma}(t)\| d t=\int_{0}^{y} \sqrt{1+4 a^{2} t^{2}+4 a b t+b^{2}} d t
$$

as new first coordinate, and zero as the second so $(y, z) \longrightarrow(L(y), 0)$.
To define the position of the points $(y, z)$ close to the parabola the first method attempted was to map them as $(y, z) \longrightarrow\left(L(y), z-\left(a y^{2}+b y+c\right)\right)$, using the vertical distance from the reference parabola as new z value. Such a solution gets increasing oscillating results the further the points are from the parabola vertex and this made the approximation process more complex.

A more refined solution would be to find the closer point $(\tilde{y}, \tilde{z})$ in the parabola and transform it as

$$
(y, z) \longrightarrow(L(\tilde{y}), \operatorname{dist}((y, z),(\tilde{y}, \tilde{z})))
$$

which keep the points in the new parameterization much flatterer.
Approximation on the newly parameterized set of points presented lower error, but those transformations inevitably change the shapes and relative dimensions of the damages. Tests performed on the re-parameterized patches showed sizable improvement in the approximation error but the damage analysis results were less reliable because of the distortion induced.

Finding $(\tilde{y}, \tilde{z})$ points requires the solution of a minimization problem. While curves different from the parabola could be used to do change the parameterization the difficulty involved grows rapidly with the curve complexity.

Reverting the points back to their original positions maintaining the differences evaluated with the approximation obtained on the transformed version could solve this problem. Further research in this direction could reduce the sensitivity of the algorithm to the edge shape, allowing simpler partitioning method to return manageable patches.

### 6.3 Conclusions

The algorithm presented uses simple mathematical and statistical tools to solve the detection and classification problem. The proposed method is adaptable to different blade models given its local nature and robust to deformation of the scanned object.

The analysis parameters are set once for each blade model and the operator only need to check the inclusion of the blade in the partitioning grid, making the procedure highly automatized. The partitioning of the blade and the separate analysis on the distinct parts makes it also highly parallelizable.

Good results were obtained on blisk blades, where all the damages can be detected with a small number of additional false positives. On worn compressor blades the FODs visually detected are mostly identified by the method proposed, with a higher number of false positives included.

The edges are the most difficult portion to analyze because of their shape, while airfoil and borders allow a high percentage of damages to be identified correctly. On those parts the algorithm results can be considered reliable.

The computational time required to perform the procedure is kept low without affecting the precision as the low degrees polynomial and splines with low number of control points show to be the best choice for the approximating function. The partitioning of the surface is usually the most time consuming part of the method.

The use of the algorithm as the only tool for FODs identification is unadvised, the missed damages on the edges and the corroded parts returned as false positives are still problematic. Ideas on how to improve the procedure have been listed and might solve those issues.

As it is the algorithm could already be used as a support tool to ease the identification of the damages. The results obtained could be visually checked to remove the false positives, fastening considerably the time necessary for the analysis.

## Appendix A

## Code

## A. 1 Main

The following functions are the one to be called to apply the algorithm to a specific (blisk) blade with splines or polynomials approximants. In both cases the results are returned as csv files in a format usable by GOM [26].

## A.1.1 blisk_main_spline

Spline approximant analysis for blisk blades.

```
import numpy as np
import scipy as sp
import trimesh as tm
from analysis_spline_main import blisk_spline__detection
from analysis__spline_main import export_gom
print('If the blade is not in the same folder as the code insert the complete path
    , otherwise press ENTER.')
path=input('Path? ')
print('Type the full name of the blisk blade, .stl included.')
name=input('Name? ')
original_mesh=tm.load_mesh(path+name)
mesh=original_mesh.copy()
damages, values=blisk_spline_detection(mesh)
export_gom(damages, path, name)
```


## A.1.2 blade_main_spline

Spline approximant analysis for worn blades.

```
import numpy as np
import scipy as sp
import trimesh as tm
from analysis_spline_main import blade_spline_detection
from analysis_spline_main import export_gom
print('If the blade is not in the same folder as the code insert the complete path
    otherwise press ENTER.')
path=input('Path? ')
print('Type the full name of the blade, . stl included.'')
name=input('Name? ')
```

```
original_mesh=tm.load_mesh(path+name)
mesh=original_mesh.copy()
damages, values=blade_spline_detection(mesh)
export_gom(damages, path, name)
```


## A.1.3 blisk_main_poly

Polynomial approximant analysis for blisk blades.

```
import numpy as np
import scipy as sp
import trimesh as tm
from analysis_poly_main import blisk_polynomial_detection
from analysis_poly_main import export_gom
print('If the blade is not in the same folder as the code insert the complete path
    , otherwise press ENTER.')
path=input('Path? ')
print('Type the full name of the blisk blade, .stl included.')
name=input('Name? ')
original_mesh=tm.load_mesh(path+name)
mesh=original_mesh.copy()
damages, values=blisk_polynomial_detection(mesh)
export_gom(damages, path, name)
```


## A.1.4 blade_main_poly

Polynomial approximant analysis for worn blades.

```
import numpy as np
import scipy as sp
import trimesh as tm
from analysis_poly_main import blade_polynomial_detection
from analysis_poly_main import export_gom
print('If the blade is not in the same folder as the code insert the complete path
    , otherwise press ENTER.')
path=input('Path? ')
print('Type the full name of the blade, .stl included.')
name=input('Name? ')
original_mesh=tm.load_mesh(path+name)
mesh=original_mesh.copy()
damages, values=blade_polynomial_detection(mesh)
export_gom(damages, path, name)
```


## A. 2 Analysis

The following functions are necessary to perform the analysis of the surface. The scripts are divided between polynomial and spline one for clarity.

## A.2.1 analysis_spline_main

Summarizing function to apply the analysis with splines approximations on the patches extracted, based on the position on the blades. Blisk blades and worn blades are treated differently
based on their shape.

```
import numpy as np
import scipy as sp
import trimesh as tm
import csv
from partition_main import blisk_partitioning
from partition_main import blade_partitioning
from partition_main import allignment
from analysis_splines import BSpline_analysis_short
#Main analysis function
def blade_spline_detection(mesh, pt=np.array([[x,x],[x,x],[x,x],[x,x]]), size=2.,
    extra=0.5, degx = [5,5,5,5,10,10,10,10], degy =[5,5,5,5,10,10,10,10], tol1
    =[0.002,0.005,0.005,0.005,0.01,0.005,0.01,0.005], tol2
    = [0.0001,0.0001,0.0001,0.0001,0.0001,0.0001,0.0001,0.0001], edges=True) :
    # INPUT
    # mesh: trimesh object, the mesh to be analyzed
    # pt: (4,2) array, corners of the partitioning grid
    # size: float, the mazimum size of a patch
    # extra: float, the amount of overlapping for adjacent patches
    # degx: (8,) list of int, number of control points along the x axis for the
    various parts of the blade
    # degy: (8,) list of int, number of control points along the y axis for the
    various parts of the blade
    # tol1: (8,) list of float, first parameter for the vaious parts of the blade
    # tol2: (8,) list of float, second parameter for the various part of the blade
    # edges: bool, if False does not divide further the edges columns in the grid
    # OUTPUT
    # damages: (n,3) array, center of the damages detected
    # values: (n,3) list, values[0]=normalized PC values for the damage, values
    [1]=[length, width, depth], values[2]=area
    damages=[]
    values=[]
    flat,top, bordo1, bordo2, edge1, edge2, edge11, edge22,tempo1,tempo2,tempo3=
    blade_partitioning(mesh,pt, size, extra, edges)
    if len(flat)>0:#analysis on the airfoil
            straightened, matrixes, centers=allignment(flat, parabole=False)
            danni, error, valori= BSplines_analysis_short(straightened,matrixes,
    centers, 2, 2, degx[0], degy[0],tol1[0], tol2[0])
        damages.extend(danni.tolist())
        values.extend(valori)
    if len(top)>0:#analysis on the top part
            straightened, matrixes, centers=allignment(top, parabole=False)
            danni, error, valori= BSplines_analysis_short(straightened,matrixes,
    centers, 2,2, degx[1], degy[1],tol1[1],tol2[1])
        damages.extend(danni.tolist())
        values.extend(valori)
    if len(bordo1)>0:#analysis on the left border
        straightened, matrixes, centers=allignment(bordo1, parabole=False)
        danni, error, valori= BSplines_analysis_short(straightened,matrixes,
    centers,2,2, degx[2], degy[2],tol1[2],tol2[2])
        damages.extend(danni.tolist())
        values.extend(valori)
    if len(bordo2)>0:#analysis on the right border
        straightened, matrixes, centers=allignment(bordo2, parabole=False)
        danni, error, valori= BSplines_analysis_short(straightened,matrixes,
    centers,2,2, degx[3], degy[3],tol1[3],tol2[3])
        damages.extend(danni.tolist())
        values.extend(valori)
    if len(edge1)>0:#analysis on the left edge (first piece)
```

```
    straightened, matrixes, centers=allignment(edge1, parabole=True)
```

    straightened, matrixes, centers=allignment(edge1, parabole=True)
    danni, error, valori= BSplines_analysis_short(straightened, matrixes,
    danni, error, valori= BSplines_analysis_short(straightened, matrixes,
    centers,2,2, degx[4], degy[4],tol1[4],tol2[4])
    centers,2,2, degx[4], degy[4],tol1[4],tol2[4])
        damages.extend(danni.tolist())
        damages.extend(danni.tolist())
        values.extend(valori)
        values.extend(valori)
    if len(edge2)>0:#analysis on the right edge (first piece)
    if len(edge2)>0:#analysis on the right edge (first piece)
        straightened, matrixes, centers=allignment(edge2, parabole=True)
        straightened, matrixes, centers=allignment(edge2, parabole=True)
        danni, error, valori= BSplines_analysis_short(straightened, matrixes,
        danni, error, valori= BSplines_analysis_short(straightened, matrixes,
    centers,2,2, degx[5], degy[5],tol1[5],tol2[5])
    centers,2,2, degx[5], degy[5],tol1[5],tol2[5])
        damages.extend(danni.tolist())
        damages.extend(danni.tolist())
        values.extend(valori)
        values.extend(valori)
    if len(edge11)>0:#analysis on the left edge (second piece)
    if len(edge11)>0:#analysis on the left edge (second piece)
        straightened, matrixes, centers=allignment(edge11, parabole=True)
        straightened, matrixes, centers=allignment(edge11, parabole=True)
        danni, error, valori= BSplines_analysis_short(straightened,matrixes,
        danni, error, valori= BSplines_analysis_short(straightened,matrixes,
    centers,2,2, degx[6], degy[6],tol1[6],tol2[6])
    centers,2,2, degx[6], degy[6],tol1[6],tol2[6])
        damages.extend(danni.tolist())
        damages.extend(danni.tolist())
        values.extend(valori)
        values.extend(valori)
    if len(edge22)>0:#analysis on the right edge (second piece)
    if len(edge22)>0:#analysis on the right edge (second piece)
        straightened, matrixes, centers=allignment(edge22, parabole=True)
        straightened, matrixes, centers=allignment(edge22, parabole=True)
        danni, error, valori= BSplines_analysis_short(straightened,matrixes,
        danni, error, valori= BSplines_analysis_short(straightened,matrixes,
    centers,2,2, degx[7], degy[7],tol1[7],tol2[7])
    centers,2,2, degx[7], degy[7],tol1[7],tol2[7])
        damages.extend(danni.tolist())
        damages.extend(danni.tolist())
        values.extend(valori)
        values.extend(valori)
    damages=np.array (damages)
    damages=np.array (damages)
    return damages, values
    return damages, values
    \#Main analysis function
\#Main analysis function
def blisk_polynomial_detection(mesh, pt=np.array([[x, x],[x,x],[x,x],[x,x]]), pt1=
def blisk_polynomial_detection(mesh, pt=np.array([[x, x],[x,x],[x,x],[x,x]]), pt1=
np.array([[x, x],[x, x],[x, x],[x,x]]), pt2=np.array([[x, x],[x, x],[x, x],[x, x]]),
np.array([[x, x],[x, x],[x, x],[x,x]]), pt2=np.array([[x, x],[x, x],[x, x],[x, x]]),
size=2., extra=0.5, degx = [5,5,5,5,10,10,10,10], degy = [5,5,5,5,10,10,10,10],
size=2., extra=0.5, degx = [5,5,5,5,10,10,10,10], degy = [5,5,5,5,10,10,10,10],
tol1 = [0.002,0.005,0.005,0.005,0.01,0.005,0.01,0.005], tol2
tol1 = [0.002,0.005,0.005,0.005,0.01,0.005,0.01,0.005], tol2
= [0.0001,0.0001,0.0001,0.0001,0.0001,0.0001,0.0001,0.0001], edges=True):
= [0.0001,0.0001,0.0001,0.0001,0.0001,0.0001,0.0001,0.0001], edges=True):
\# INPUT
\# INPUT
\# mesh: trimesh object, the mesh to be analyzed
\# mesh: trimesh object, the mesh to be analyzed
\# pt: (4,2) array, corners of the partitioning grid
\# pt: (4,2) array, corners of the partitioning grid
\# pt1: (4,2) array, corners of the partitioning grid for the lower part
\# pt1: (4,2) array, corners of the partitioning grid for the lower part
\# pt2: (4,2) array, corners of the partitioning grid for the upper part
\# pt2: (4,2) array, corners of the partitioning grid for the upper part
\# size: float, the mazimum size of a patch
\# size: float, the mazimum size of a patch
\# extra: float, the amount of overlapping for adjacent patches
\# extra: float, the amount of overlapping for adjacent patches
\# degx: (8,) list of int, degrees of the polynomials along the x axis for the
\# degx: (8,) list of int, degrees of the polynomials along the x axis for the
various parts of the blade
various parts of the blade
\# degy: (8,) list of int, degrees of the polynomials along the y axis for the
\# degy: (8,) list of int, degrees of the polynomials along the y axis for the
various parts of the blade
various parts of the blade
\# tol1: (8,) list of float, first parameter for the vaious parts of the blade
\# tol1: (8,) list of float, first parameter for the vaious parts of the blade
\# tol2: (8,) list of float, second parameter for the various part of the blade
\# tol2: (8,) list of float, second parameter for the various part of the blade
\# edges: bool, if False does not divide further the edges columns in the grid
\# edges: bool, if False does not divide further the edges columns in the grid
\# OUTPUT
\# OUTPUT
\# damages: (n,3) array, center of the damages detected.
\# damages: (n,3) array, center of the damages detected.
\# values: (n,3) list, values[0]=normalized PC values for the damage, values
\# values: (n,3) list, values[0]=normalized PC values for the damage, values
[1]=[length, width, depth], values[2]=area
[1]=[length, width, depth], values[2]=area
damages=[]
damages=[]
values=[]
values=[]
flat,top, bordo1, bordo2, edge1, edge2, edge11, edge22,tempo1, tempo2,tempo3=
flat,top, bordo1, bordo2, edge1, edge2, edge11, edge22,tempo1, tempo2,tempo3=
blisk_partitioning(mesh, pt, size, extra, pt1, pt2, edges)
blisk_partitioning(mesh, pt, size, extra, pt1, pt2, edges)
if len(flat)>0:\#analysis on the airfoil
if len(flat)>0:\#analysis on the airfoil
straightened, matrixes, centers=allignment(flat, parabole=False)
straightened, matrixes, centers=allignment(flat, parabole=False)
danni, error, valori= BSplines_analysis_short(straightened,matrixes,
danni, error, valori= BSplines_analysis_short(straightened,matrixes,
centers, 2, 2, degx[0], degy[0], tol1[0], tol2[0])
centers, 2, 2, degx[0], degy[0], tol1[0], tol2[0])
damages.extend(danni.tolist())
damages.extend(danni.tolist())
values.extend(valori)

```
    values.extend(valori)
```

```
    if len(top)>0:#analysis on the top part
        straightened, matrixes, centers=allignment(top, parabole=False)
        danni, error, valori= BSplines_analysis_short(straightened,matrixes,
    centers, 2, 2, degx[1], degy[1],tol1[1],tol2[1])
        damages.extend(danni.tolist())
        values.extend(valori)
    if len(bordo1)>0:#analysis on the left border
        straightened, matrixes, centers=allignment(bordo1, parabole=False)
        danni, error, valori= BSplines_analysis_short(straightened,matrixes,
    centers,2,2, degx[2], degy[2],tol1[2],tol2[2])
        damages.extend(danni.tolist())
        values.extend(valori)
    if len(bordo2)>0:#analysis on the right border
        straightened, matrixes, centers=allignment(bordo2, parabole=False)
        danni, error, valori= BSplines_analysis_short(straightened,matrixes,
    centers, 2, 2, degx[3], degy[3],tol1[3],tol2[3])
        damages.extend(danni.tolist())
        values.extend(valori)
    if len(edge1)>0:#analysis on the left edge (first piece)
        straightened, matrixes, centers=allignment(edge1, parabole=True)
        danni, error, valori= BSplines_analysis_short(straightened,matrixes,
    centers, 2, 2, degx[4], degy[4],tol1[4],tol2[4])
        damages.extend(danni.tolist())
        values.extend(valori)
    if len(edge2)>0:#analysis on the right edge (first piece)
        straightened, matrixes, centers=allignment(edge2, parabole=True)
        danni, error, valori= BSplines_analysis_short(straightened,matrixes,
    centers,2,2, degx[5], degy[5],tol1[5],tol2[5])
        damages.extend(danni.tolist())
        values.extend(valori)
    if len(edge11)>0:#analysis on the left edge (second piece)
        straightened, matrixes, centers=allignment(edge11, parabole=True)
        danni, error, valori= BSplines_analysis_short(straightened,matrixes,
    centers,2,2, degx[6], degy[6],tol1[6],tol2[6])
        damages.extend(danni.tolist())
        values.extend(valori)
    if len(edge22)>0:#analysis on the right edge (second piece)
        straightened, matrixes, centers=allignment(edge22, parabole=True)
        danni, error, valori= BSplines_analysis_short(straightened,matrixes,
    centers, 2,2, degx[7], degy[7],tol1[7],tol2[7])
        damages.extend(danni.tolist())
        values.extend(valori)
    damages=np.array(damages)
    return damages, values
#Utility function to save the damages location in a format importable by GOM
def export_gom(found, path, name):
    # INPUT
    # found: (n,3) array, coordinates of the damages locations
    # path: string, directory the file should be saved in
    # name: string, blade name, .stl included
    # OUTPUT
    # a csv file, named ''name', _findings.csv, with the damages location, readable
    by GOM
    csv=open(path+name[:-3]+'__findings.csv', "w")
    for i in range(len(found)):
        csv.write('Found'+str(i))
        for j in found[i]:
            number_int=str(int(j))
```

```
        number_dec = str(abs(j-int(j)))[2:]
        if number_int=='0', and j<0:
            number_int='-'+number_int
        csv.write('','+number_int+','+number_dec)
        csv.write('\n')
csv.close()
```


## A.2.2 analysis_spline

Subroutines necessary to build the spline approximation and perform the analysis.

```
import numpy as np
import scipy as sp
import time
import matplotlib
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import networkx as nx
import trimesh as tm
from splipy import BSplineBasis, Curve, Surface
from utilities import sub_mesh
from ulilities import estimate_matrix
from utilities import homograpy
#Utility function to evaluate the derivative of the B-splines functions
def integranda(u,basi,i,j, deriv):
    #INPUT
    # u: (2,) array, coordinates of one point in the unit square where the
    derivative should be evaluated
    # basi: B-splines basis function objects
    # i: int, knot index along the x axis
    # j: int, knot index along the y axis
    # deriv: int, order of the derivative to be evaluate, 0 indicates the original
        function value
    # OUTPUT
    # float, derivative value in the (i,j) knot evaluated on u
    b=basi.evaluate(u, deriv)
    return b[0,i]*b[0,j]
#This function determines the values of the control points necessary to define the
    spline approximant.
def least_square_fit_basic(mesh, bases, u, usabili, bordi=False, l=0):
    #Find the control points that allow for the best approximation of the points
    given
    ######Input
    # x: (n,3) array, the points considered, in their phisical coordinates
    # bases: BSpilneBasis object, the basis functions for the splines
    # u: (2,n) array, the points considered, with the parametrized (u,v) ~}=(x,y
    coordinates in the unit square.
    # l: float, if positive add the thin blade spline smoothing funcion with
    weight l
    ##|||||Output
    # Surface object, with the control points found and the basis functions given
    ###Observation: the shape of the control points array get internally adjusted
    by surface from a [nx*ny,3] to a [ny,nx,3] form if not raw=True
    x=mesh.triangles_center[usabili]
    pesi=np.zeros(len(usabili))
    #this part applies different weight to the points based on the normal
    direction. The more a triangular face is vertically inclined the higher its
    weight in the approximation.
```

```
if bordi:
    #The reference direction is chosen as the mean value of a subset of faces
normals close to the center of the mesh
    centro=np.mean(x, axis=0)
    #this is the central one, its neighborogh is the subset
    faccia=np.argmin(np.linalg.norm(mesh.triangles_center-centro, axis=1))
    #Build the adjacency graph
    graph = nx.Graph()#
    graph.add_edges_from(mesh.face_adjacency)#
    indici=[]
    ll=list(graph.neighbors(faccia))
    indici.extend(11)
    indexes=np.sort(np.unique(indici))
    ####
    #If we want to use a larger number of faces to build the normal we have to
    repeat this part
    for i in indexes:
        ll=list(graph.neighbors(i))
        indici.extend(ll)
    indici=np.array(indici)
    indici=np.append(indici, indexes)
    indexes=np.sort(np.unique(indici))
    ###
    #The reference normal is the mean value of the selected faces ones
    normale=np.mean(mesh.face_normals[indexes], axis=0)
    #define the weights
    for i in np.arange(len(usabili)):
        scalare=np.dot(normale, mesh.face_normals[usabili[i]])
        pesi[i]=1+(1-scalare)*500#this can be modified to change the weight
    values
    #shows the weights values
    fig1 = plt.gcf()
    ax1 = fig1.add_subplot(1, 2,1, projection='3d')
    ax1.scatter( x[:,0], x[:, 1], pesi,c='b')
    #shows the patch shape
    ax2 = fig1.add_subplot(1, 2, 2, projection='3d')
    ax2.scatter( x[:,0], x[:,1], x[:, 2],c='b')
    plt.show(block=False)
    #evaluates the 1D basis over the data points(in [0,1]^2)
    N_all1=np.array (bases[0](u[0]))
    N_all2=np.array(bases[1](u[1]))
    #build the matrix with the 2D basis values
    U=N_all1[0,:]
    V=N_all2[0,:]
    A=np.dot(U.reshape( - 1, 1),V.reshape (1, -1)).reshape (1, - ) # #* pesi [0]
    for i in range(1,len(N_all1)):
    U=N_all1[i,:]
    V=N_all2[i,:]
    M=np.dot(U.reshape( - 1,1),V.reshape (1, -1)).reshape(1, - 1)#* pesi [ i ]
    A=np.vstack((A,M))
    print('cond. numb of A=',np.linalg.cond(A))
    A2=A.copy ()
    #this part adds a smoothing parameters of weight l.
    if l>0:
    X0=np.zeros((len(U), len(U)))
    X1=X0. copy ()
    X2=X0. copy ()
    Y0=np.zeros((len(V), len(V)))
    Y1=Y0. copy ()
    Y2=Y0. copy()
```

```
    print(bases[0]== bases[1])
    for i in np.arange(len(U)):
        for r in np.arange(len(U)):
        X0[i, r]=sp.integrate.quad(integranda, 0, 1, args=(bases[0],i, r, 0) ) [0]
        X1[i,r]=sp.integrate.quad(integranda, 0, 1, args=(bases [0], i, r, 1)) [0]
        X2[i,r]=sp.integrate.quad(integranda, 0,1, args=(bases[0],i,r, 2) ) [0]
    for j in np.arange(len(V)):
    for s in np.arange(len(V)):
        Y0[j, s]=sp.integrate.quad(integranda, 0, 1, args=(bases [1],j, s,0) ) [0]
        Y1[j, s]=sp.integrate.quad(integranda, 0, 1, args=(bases [1],j, s, 1) ) [0]
        Y2[j, s]=sp.integrate.quad(integranda, 0, 1, args=(bases[1], j, s, 2)) [0]
    M=np.zeros((len(U)*len(V), len(U)*len(V)))
    k=0
    for i in np.arange(len(U)):
        for j in np.arange(len(V)):
        a=np.dot(X2[i].reshape( - 1, 1), Y0[j].reshape(1, - 1)).reshape(1, - 1)
        b=np.dot(X1[i].reshape( - 1, 1), Y1[j].reshape(1, - 1)).reshape(1, - 1)
        c=np.dot(X0[i ]. reshape( - 1, 1),Y2[j].reshape(1, - 1)).reshape(1, - 1)
        M[k,:]=a#+2*b#+c
        k+=1
    #to add the smoothing factor is necessary to solve a sistem with higher
    conditioning number, this shows it and the effect of the smoothing factor on
    it.
    print('cond. numb of A.T*A=',np.linalg. cond(np.dot (A.T,A)))
    print('cond. numb of A.T*A+l*E=',np.linalg. cond(np.dot (A.T,A)+l*M))
    cpxy,_,_,_=np.linalg.lstsq(A, x[:,:2])
    cpz=nр.linalg. solve(np.dot(A.T,A)+l*M, np.dot (A2.T, x [:, 2]. reshape(-1,1))).
reshape (1, -1)
    controlpoints=np.vstack((cpxy.T, cpz)).T
    else: #if necessary adds the weights
    if bordi:
        A2 = A * np.sqrt(pesi[:, np.newaxis])
        x[:,2] = x[:, 2] * np.sqrt(pesi)
        print('cond. numb of weighted A=',np.linalg.cond(A2))
    #otherwise the control points are evaluated using the least square method
    cpxy,_,_,_=np.linalg.lstsq(A, x[:, : 2])
    cpz,_,_,_=np.linalg.lstsq(A2, x[:, 2])
    controlpoints=np.vstack((cpxy.T, cpz)).T
    #print(controlpoints)
    #if len(U)!=len(V):
    cp=[]
    #the control points have to be reshaped to be used by splipy
    for i in range(len(V)):
    cp.append(controlpoints[np.arange(i, len(controlpoints), len(V))])
    controlpoints=np.asarray (cp).transpose((1, 0, 2))
    return Surface(bases [0], bases [1], controlpoints, raw=True)
    #return Surface(bases[0], bases[1], np.array(controlpoints))
#Potential damage detector
def parameters_centroids1_short(model, scan, tol):
    # INPUT
    # model: (n,3) array, coordinates of the points of the model with the same
    parametrization of the data points.
    # scan: Trimesh object, the surface analized
    # tol: costant value, threshold for differences, called first parameter or
    first threshold in the thesis
    # visual: bool, if True show graphic rapresentation of various itermediary
    results.
    # OUTPUT
```

```
# damaged_locations: list of trimesh objects, each one represent a dmaged
```


# damaged_locations: list of trimesh objects, each one represent a dmaged

location
location

# root_mean_diff: float, root mean square error between patch and model

# root_mean_diff: float, root mean square error between patch and model

    diff=model[:, 2] - scan.triangles_center [: , 2]
    diff=model[:, 2] - scan.triangles_center [: , 2]
    root_mean_diff=np.sqrt(sum(diff}**2)/len(scan.faces))
    root_mean_diff=np.sqrt(sum(diff}**2)/len(scan.faces))
    diff_filtered=np.zeros(len(diff))
    diff_filtered=np.zeros(len(diff))
    indexes=[]
    indexes=[]
    ##this does not consider the difference between holes and protrusion.
    ##this does not consider the difference between holes and protrusion.
    for n in range(len(diff)):
    for n in range(len(diff)):
        if abs(diff[n])>tol:
        if abs(diff[n])>tol:
        indexes.append(n)
        indexes.append(n)
        diff_filtered[n]=np.sign(diff[n])*(abs(diff[n])-tol)#I'm not sure the
        diff_filtered[n]=np.sign(diff[n])*(abs(diff[n])-tol)#I'm not sure the
    sign is necessary
sign is necessary
if indexes==[]:
if indexes==[]:
return indexes , root_mean_diff
return indexes , root_mean_diff
diff_pt=scan.triangles_center[:,: 2][indexes]
diff_pt=scan.triangles_center[:,: 2][indexes]
indexes=np.array(indexes)
indexes=np.array(indexes)
\#we create a list of points with damages and a figure to show them
\#we create a list of points with damages and a figure to show them
diff_pt_value=np.vstack((diff_pt.T, diff_filtered[indexes])).T
diff_pt_value=np.vstack((diff_pt.T, diff_filtered[indexes])).T
mesh1=scan.copy()
mesh1=scan.copy()
mesh1.update_faces(indexes)
mesh1.update_faces(indexes)
graph = nx.Graph()
graph = nx.Graph()
graph.add_nodes_from(np.arange(len(scan.faces)))
graph.add_nodes_from(np.arange(len(scan.faces)))
graph.add_edges_from(mesh1.face_adjacency)
graph.add_edges_from(mesh1.face_adjacency)
groups = nx.connected_components(graph)\#
groups = nx.connected_components(graph)\#
gruppi=np.array(list(groups))
gruppi=np.array(list(groups))
l=[]
l=[]
singoletti=[]
singoletti=[]
usati=0
usati=0
n=0
n=0
while n<len(gruppi) and usati<len(indexes):
while n<len(gruppi) and usati<len(indexes):
i=gruppi[n]
i=gruppi[n]
n+=1
n+=1
usati+=len(list (i))
usati+=len(list (i))
if len(i)}>1\mathrm{ :
if len(i)}>1\mathrm{ :
l.append(indexes[list(i)])
l.append(indexes[list(i)])
else:
else:
singoletti.extend(indexes[list(i)])
singoletti.extend(indexes[list(i)])
damaged_locations=[]
damaged_locations=[]
graph = nx.Graph()
graph = nx.Graph()
graph.add_nodes_from(np.arange(len(scan.faces)))
graph.add_nodes_from(np.arange(len(scan.faces)))
graph.add_edges_from(scan.face_adjacency)
graph.add_edges_from(scan.face_adjacency)
for j in \overline{l}
for j in \overline{l}
indici=[]
indici=[]
for i in j:
for i in j:
ll=list(graph[i])
ll=list(graph[i])
indici.extend(ll)
indici.extend(ll)
j=np.array(indici)
j=np.array(indici)
indici=[]
indici=[]
for i in j:
for i in j:
ll=list(graph[i])
ll=list(graph[i])
indici.extend(ll)
indici.extend(ll)
indici=np.array(indici)
indici=np.array(indici)
j=np.unique(indici)
j=np.unique(indici)
new_mesh=sub_mesh(scan,j)

```
        new_mesh=sub_mesh(scan,j)
```

```
        damaged_locations.append(new_mesh)
    for j in singoletti:
        j=graph.neighbors(j)
        if len(j)>0:
        for k in range(5):
            indici=[]
                for i in j:
                ll=list(graph[i])
                indici.extend(ll)
            indici=np.array(indici)
            j=np.unique(np.append(indici, j))
        new_mesh=sub_mesh(scan,j)
        damaged_loc\overline{ations.append(new_mesh)}
    return damaged_locations , root_mean_diff
#PCA evaluator
def parameters_centroids2_short(model,scan) :
    # INPUT
    # model: (n,3) array, coordinates of the points of the model with the same
    parametrization of the data points. Damage only.
    # scan: Trimesh object, the surface analized. Damage only.
    # OUTPUT
    # S: (3,) array, PC values from the covariance matrix on 3D points
    # V: (3,3) matrix, PC directions from the covariance matrix on 3D points
    #re-evaluate the differences
    diff=model[:, 2] - scan.triangles_center [:, 2]
    #if less than 2 point are present on the damage no meaningful analysis can be
    performed. the process is then interrupted and returns only zeros.
    if len(diff)<2:
        print("The damage is too small here")
        a=np.array ([0,0,0])
        b=np}\cdot\operatorname{array}([0,0,0],[0,0,0],[0,0,0]
        return a,b
    #build a new set of point with the same xy coordinates and the differences as
    z values
    diff_pt=scan.triangles_center [:, :2]
    diff_pt_value=np.vstac\overline{k}}((\mp@subsup{d}{iff_pt.T, diff)).T}{
    #build the covariance matrix
    mean=np.mean(diff_pt_value, axis=0)
    covariance=np.dot((diff_pt_value-mean).T, diff_pt_value-mean)
    #Use the SVD decomposition to compute the PC values and vectors
    U,S,V=np.linalg.svd(covariance)
    #print(S1/np.linalg.norm(S1))
    #Return the principal values and vectors for covariance and correlation
    matrixes
    return S, V
def BSplines analysis short(dritte, matrici, centri, degreeu, degreev, ncpu, ncpv, tol1
    =0.005, tol2 = 0.0001, shape=10, tol_shape = [0.0075,0.15], border = [0,0,0]):
    # INPUT
    # dritte: (n,) array of trimesh objects, each mesh is centered in the origin
    and oriented as described in the thesis.
    # matrici: (n,) array of (3,3) matrixes, each one is the rotation necessary to
    revert the respective mesh to its original orientation
    # centri: (n,) array of (3,) vectors, the translation necessary to move the
    respective mesh to its original position
    # degreeu: int, degree of the splines along the x axis
```

```
262
263
264
265
266
267
268
```


# degreev: int, degree of the splines along the y axis

```
# degreev: int, degree of the splines along the y axis
# ncpu: int, number of control points along the x axis
# ncpu: int, number of control points along the x axis
# ncpv: int, number of control points along the y axis
# ncpv: int, number of control points along the y axis
# tol1: float, first threshold or parameter, as described in the thesis
# tol1: float, first threshold or parameter, as described in the thesis
# tol2: float, second threshold or parameter, as described in the thesis
# tol2: float, second threshold or parameter, as described in the thesis
# shape: int, maximum lenght/width ratio
# shape: int, maximum lenght/width ratio
# tol_shape: (2,) list, the minimum depth and minimum width of an acceptable
# tol_shape: (2,) list, the minimum depth and minimum width of an acceptable
damage
damage
# border: (3,) array, the amount the corners used for the homograpy should be
# border: (3,) array, the amount the corners used for the homograpy should be
widened along the x,y and z directions.
widened along the x,y and z directions.
# OUTPUT
# OUTPUT
# danni: (m,3) array, coordinates of the centers of the damages detected
# danni: (m,3) array, coordinates of the centers of the damages detected
# error: (n,2) array, contains the RMSE and the number of faces for each patch
# error: (n,2) array, contains the RMSE and the number of faces for each patch
# valori: (m,3) array, valori[0]= normalized PC values for the damages, valori
# valori: (m,3) array, valori[0]= normalized PC values for the damages, valori
[1]=[length, width, depth], valori[2]=area
[1]=[length, width, depth], valori[2]=area
start_analysis_time = time.time()
start_analysis_time = time.time()
numero=1
numero=1
danni=[]
danni=[]
valori=[]
valori=[]
faces_number =[]
faces_number =[]
mean_error =[]
mean_error =[]
#####This part build the B-spline function necessary to define the approximant
#####This part build the B-spline function necessary to define the approximant
p=degreeu+1
p=degreeu+1
n=ncpu
n=ncpu
#we define the knots along x~u direction
#we define the knots along x~u direction
knot1 = [0] * p + list(range(1, n - p + 1)) + [n - p + 1] * p
knot1 = [0] * p + list(range(1, n - p + 1)) + [n - p + 1] * p
knot1=np.asarray(knot1)/knot1[-1]
knot1=np.asarray(knot1)/knot1[-1]
print(knot1)
print(knot1)
#and build the BSpline basis over them
#and build the BSpline basis over them
basis1 = BSplineBasis(p, knot1)
basis1 = BSplineBasis(p, knot1)
#again along y~v
#again along y~v
p=degreev +1
p=degreev +1
n=ncpv
n=ncpv
knot2 = [0] * p + list(range(1, n - p + 1)) + [n - p + 1] * p
knot2 = [0] * p + list(range(1, n - p + 1)) + [n - p + 1] * p
knot2=np.asarray(knot2)/knot2[-1]
knot2=np.asarray(knot2)/knot2[-1]
print(knot2)
print(knot2)
basis2 = BSplineBasis(p, knot2)
basis2 = BSplineBasis(p, knot2)
###
###
#The analisis is performed on each patch singularly
#The analisis is performed on each patch singularly
for m in dritte:
for m in dritte:
    matrice=matrici[numero-1]
    matrice=matrici[numero-1]
    centro=centri [numero-1]
    centro=centri [numero-1]
    faces_number.append(len(m.faces))
    faces_number.append(len(m.faces))
    print("##||||||||||||||||||")
    print("##||||||||||||||||||")
    print('Mesh number %d' %numero)
    print('Mesh number %d' %numero)
    print("##|||||||||||||||||")
    print("##|||||||||||||||||")
    print("Centered in ",centro)
    print("Centered in ",centro)
    numero+=1
    numero+=1
    ##|||||||||||||||||||||||
    ##|||||||||||||||||||||||
    #we divide the coordinates of the points to better use them
    #we divide the coordinates of the points to better use them
    centers_xy=m.triangles_center [:,:2]
    centers_xy=m.triangles_center [:,:2]
    centers_z=m.triangles_center[:, 2]
    centers_z=m.triangles_center[:, 2]
    #We find the x and y values of the rectangle that cover the triangles
    #We find the x and y values of the rectangle that cover the triangles
centers of the mesh, the addition are useful to better contain the points
centers of the mesh, the addition are useful to better contain the points
    sotto=min(centers_xy[:,1])-0.1#lower point along x
    sotto=min(centers_xy[:,1])-0.1#lower point along x
    sopra=max(centers_xy[:,1]) +0.1#higher point along x
    sopra=max(centers_xy[:,1]) +0.1#higher point along x
    sinistra=min(centers_xy[:,0]) - 0.1#lower point along y
    sinistra=min(centers_xy[:,0]) - 0.1#lower point along y
    destra=max(centers_xy[:,0])+0.1#higher point along y
    destra=max(centers_xy[:,0])+0.1#higher point along y
    spigoli=np.array([[sinistra, sotto],[ sinistra, sopra],[destra, sopra],[destra
    spigoli=np.array([[sinistra, sotto],[ sinistra, sopra],[destra, sopra],[destra
    ,sotto]])#corners, clockwise from the bottom left one
```

    ,sotto]])#corners, clockwise from the bottom left one
    ```
```

matrice1=estimate_matrix (np.asarray ([[0, 0],[0, 1], [1, 1],[1, 0]]), spigoli) \# transformation from unit square to rectangle
matrice2=estimate_matrix (spigoli, np.asarray ([[0, 0], [0, 1],[1, 1],[1, 0]]) ) \# transformation from rectangle to unit square
\#We apply an homograpy transformation to bring all the points in the unit square(with respect to the x and y coord)
inside1=homograpy (matrice 2, m.triangles_center)
\#we check if some point is still out of the unit square. Those points are impossible to be modeled with the spline function and will be ignored. They are usually non existant and being them on the edges they are considered in at least another mesh.
fuori=0
usabili=[]
inutilizzabili=[]
$\mathrm{n}=0$
for $i$ in inside1:
if any $(\mathrm{i}<0)$ or any $(\mathrm{i}>1): \#$ condition to determine if the point is outside the unit square

$$
\text { fuori }+=1
$$

inutilizzabili.append(n)
else:
usabili.append(n)
$\mathrm{n}+=1$
if fuori>0: print("Punti fuori dai CP \%d" \%fuori)
\#Some action could be performed to include these external point, at the moment we just enlarge the starting zone with 'border'. We now evaluate the bspline on the transformed coordinates and find the control points to define the spline approximant
spline=least_square_fit_basic (m, [basis1, basis2], inside1[usabili].T, usabili )\#find the control points values
punti_spline=spline (inside1[usabili, 0], inside1[usabili, 1], tensor=False) \#
evaluate the spline function on the re-parameterized points inside1
if err_visual:
plot_3D_surface(spline,m.triangles_center, m.triangles_center)
\#spline_points are the same as punti but with the original (x,y) coord
spline_points=np.vstack ((m.triangles_center[usabili,:2].T, punti[:, 2]) ).T
\#we apply parameter_centroids1
lista, medio=parameters_centroids1_short(spline_points, sub_mesh(m, usabili), tol1)
\#we add the new root mean square error to the list
mean_error. append (medio)
\#if there are potential damages they are analyzed singularly
if len (lista) $>0$ :
print ("\#\#")
for $i$ in lista:
\#we repeat the same process but only with the points of one
damaged area
\#square give us the coord in the unit square, with the same
trasformation as before
square=homograpy (matrice2, i.triangles_center)
$z_{\text {_ }}$ values=spline (square $[:, 0]$, square $[:, \overline{1}]$, tensor=False) $[:, 2]$
\#We evaluate the BSplines on the points in square
spl_pts=np.asarray ([i.triangles_center[:, 0], i.triangles_center
[: , 1], z_values]).T
\#we apply parameters_centroids2
S_cov, V_cov=parameters_centroids2_short (spl_pts, i)
\#we normalize the vector of the PC values

```
```

            parametro=S_cov/np.linalg.norm(S_cov)
            #and find the center of the damage
            mean=np.mean(i.triangles_center, axis=0)
            original=np.dot(matrice, mean)[0]+ centro[0]
            #print some useful parameters
            print(" ")
            print("Center of the damage=",original)
            print("Un-normalized parameter=",S_cov)
            print("Considered parameters= " ,parametro)
            print("Linearity param=",(S_cov[0]-S_cov[1])/S_cov[0])
            print("Planarity param=",(S_cov[1]-S_cov[2])/S_cov[0])
            print("Sfericity param=",S_cov[2]/S_cov[0])
            print("Anisotropy=",(S_cov[0]-S_cov[2])/S_cov[0])
            print("Eighenentropy=",-S_cov[0]*np.log(S_cov[0])-S_cov[1]*np.log(
    S_cov[1])-S_cov[2]*np.log(S_cov[2]))
print("Curvature change=",S_cov[2]/(S_cov[0]+S_cov[1]+S_cov[2]))
centri_i=i.vertices -np.mean(i.vertices, axis=0)
ruotati=np.asarray(np.dot(V_cov, centri_i.T))
\#compute the damage dimensions
lunghezza=max(ruotati[0,:])-min(ruotati[0,:])\#lenght
larghezza=max(ruotati[1,:]) -min(ruotati[1,:])\#width
profondita=max(ruotati[2,:])-min(ruotati[2,:])\#depth
print('Max distances=',[lunghezza,larghezza, profondita])
\#evaluate the area and volume of the damage as the one of the
convex hull containing it
ruotati=ruotati.T
hull2D=sp.spatial.ConvexHull(ruotati [:,:2])\#using only the first
two coordinates we obtain the area
area=hull2D.volume
\#print("Area of the triangles in the convex hull =",hull2D.volume)
\#hull3D=sp.spatial.ConvexHull(ruotati[:,:])\#using all the
coordinates allow the volume to be extracted
\#print("Volume of the 3D convex hull =",hull3D.volume)
\#if the 3rd normalized PC value is higher than tol2 and the depth
is higher than tol_shape[0] the analysis continues
if parametro[2]>=tol2 and profondita>tol_shape[0]:
\#this second control compares first and second PC values to
filter out the long and thin ones. It also requres a minimum width of the
damage.
if larghezza>(lunghezza/shape) and larghezza>=tol_shape[1]:
print("Is acceptable")
valori.append([S_cov/np.linalg.norm(S_cov),[lunghezza,
larghezza, profondital, areal)
danni.append(original)
else:\# if is too long and thin is discarded
print("Is too scratch-like")
\#otherwise is a faux positive
else:
print("Is not acceptable")
danni=np.asarray(danni)
error=[mean_error, faces_number]
print("Total root mean square error=",np.sqrt(np.dot(np.asarray(error[0]) **2,
np.asarray(error[1]))/sum(error[1])))
return danni, error, valori

```

\section*{A.2.3 analysis_poly_main}

Summarizing function to apply the analysis with polynomial approximations on the patches extracted, based on the position on the blades. Blisk blades and worn blades are treated differ-
ently based on their shape.
```

import numpy as np
import scipy as sp
import trimesh as tm
import csv
from partition_main import blisk_partitioning
from partition_main import allignment
from analysis_poly import polynomial_analysis_short
\#Main analysis function
def blisk_polynomial_detection(mesh, pt=np.array([[x,x],[x,x],[x,x],[x,x]]), pt1=
np.array([[x, x],[x,x],[x, x],[x,x]]), pt2=np.array([[x, x],[x,x][x,x],[x, x]]),
size=2., extra=0.5, degx = [5,5,5,5,10,10,10,10], degy = [5,5,5,5,10,10,10,10],
tol1 = [0.002,0.005,0.005,0.005,0.01,0.005,0.01,0.005], tol2
=[0.0001,0.0001,0.0001,0.0001,0.0001,0.0001,0.0001,0.0001], edges=True):
\# INPUT
\# mesh: trimesh object, the mesh to be analyzed
\# pt: (4,2) array, corners of the partitioning grid
\# pt1: (4,2) array, corners of the partitioning grid for the lower part
\# pt2: (4,2) array, corners of the partitioning grid for the upper part
\# size: float, the mazimum size of a patch
\# extra: float, the amount of overlapping for adjacent patches
\# degx: (8,) list of int, degrees of the polynomials along the x axis for the
various parts of the blade
\# degy: (8,) list of int, degrees of the polynomials along the y axis for the
various parts of the blade
\# tol1: (8,) list of float, first parameter for the vaious parts of the blade
\# tol2: (8,) list of float, second parameter for the various part of the blade
\# edges: bool, if False does not divide further the edges columns in the grid
\# OUTPUT
\# damages: (n,3) array, center of the damages detected.
\# values: (n,3) list, values[0]= normalized PC values for the damage, values
[1]=[length, width, depth], values[2]= area
damages =[]
values=[]
flat,top,bordo1, bordo2, edge1, edge2, edge11, edge22, tempo1,tempo2, tempo3=
blisk_partitioning(mesh,pt, size, extra, pt1, pt2, edges)
if len(flat)>0:\#analysis on the airfoil
straightened, matrixes, centers=allignment(flat, parabole=False)
danni, error, valori= polynomial_analysis_short(straightened,matrixes,
centers, degx[0], degy[0],tol1[0],tol2[0], shape=True)
damages.extend(danni.tolist())
values.extend(valori)
if len(top)>0:\#analysis on the top part
straightened, matrixes, centers=allignment(top, parabole=False)
danni, error, valori= polynomial_analysis_short(straightened,matrixes,
centers,degx[1],degy[1],tol1[1],tol2[1], shape=True)
damages.extend(danni.tolist())
values.extend(valori)
if len(bordo1)>0:\#analysis on the left border
straightened, matrixes, centers=allignment(bordo1, parabole=False)
danni, error, valori= polynomial_analysis_short(straightened,matrixes,
centers,degx[2], degy[2],tol1[2],tol2[2], shape=True)
damages.extend(danni.tolist())
values.extend(valori)
if len(bordo2)>0:\#analysis on the right border
straightened, matrixes, centers=allignment(bordo2, parabole=False)
danni, error, valori= polynomial_analysis_short(straightened,matrixes,
centers, degx[3],degy[3],tol1[3],tol2[3], shape=True)

```
```

        damages.extend(danni.tolist())
    ```
        damages.extend(danni.tolist())
        values.extend(valori)
        values.extend(valori)
    if len(edge1)>0:#analysis on the left edge (first piece)
    if len(edge1)>0:#analysis on the left edge (first piece)
        straightened, matrixes, centers=allignment(edge1, parabole=True)
        straightened, matrixes, centers=allignment(edge1, parabole=True)
        danni, error, valori= polynomial_analysis_short(straightened, matrixes,
        danni, error, valori= polynomial_analysis_short(straightened, matrixes,
    centers, degx [4], degy[4], tol1 [4], tol2[4], shape=True)
    centers, degx [4], degy[4], tol1 [4], tol2[4], shape=True)
        damages.extend(danni.tolist ())
        damages.extend(danni.tolist ())
        values.extend(valori)
        values.extend(valori)
    if len(edge2)>0:#analysis on the right edge (first piece)
    if len(edge2)>0:#analysis on the right edge (first piece)
        straightened, matrixes, centers=allignment(edge2, parabole=True)
        straightened, matrixes, centers=allignment(edge2, parabole=True)
        danni, error, valori= polynomial_analysis_short(straightened, matrixes,
        danni, error, valori= polynomial_analysis_short(straightened, matrixes,
    centers, degx[5], degy[5], tol1 [5], tol2[5], shape=True)
    centers, degx[5], degy[5], tol1 [5], tol2[5], shape=True)
        damages.extend(danni.tolist ())
        damages.extend(danni.tolist ())
        values.extend(valori)
        values.extend(valori)
    if len(edge11)>0:#analysis on the left edge (second piece)
    if len(edge11)>0:#analysis on the left edge (second piece)
        straightened, matrixes, centers=allignment(edge11, parabole=True)
        straightened, matrixes, centers=allignment(edge11, parabole=True)
        danni, error, valori= polynomial_analysis_short(straightened, matrixes,
        danni, error, valori= polynomial_analysis_short(straightened, matrixes,
    centers, degx[6], degy[6], tol1 [6], tol2[6], shape=True)
    centers, degx[6], degy[6], tol1 [6], tol2[6], shape=True)
        damages.extend(danni.tolist ())
        damages.extend(danni.tolist ())
        values.extend(valori)
        values.extend(valori)
    if len(edge22)>0:#analysis on the right edge (second piece)
    if len(edge22)>0:#analysis on the right edge (second piece)
        straightened, matrixes, centers=allignment(edge22, parabole=True)
        straightened, matrixes, centers=allignment(edge22, parabole=True)
        danni, error, valori= polynomial_analysis_short(straightened, matrixes,
        danni, error, valori= polynomial_analysis_short(straightened, matrixes,
    centers, degx[7], degy[7], tol1 [7], tol2[7], shape=True)
    centers, degx[7], degy[7], tol1 [7], tol2[7], shape=True)
        damages.extend(danni.tolist ())
        damages.extend(danni.tolist ())
        values.extend(valori)
        values.extend(valori)
    damages=np.array(damages)
    damages=np.array(damages)
    return damages, values
    return damages, values
#Main analysis function
#Main analysis function
def blades_polynomial_detection(mesh, pt=np.array ([[x, x],[x, x][x, x],[x, x]]), size
def blades_polynomial_detection(mesh, pt=np.array ([[x, x],[x, x][x, x],[x, x]]), size
    =5., extra=2., degx = [5,5,5,5,10,10,10,10], degy = [5,5,5,5,10,10,10,10], tol1
    =5., extra=2., degx = [5,5,5,5,10,10,10,10], degy = [5,5,5,5,10,10,10,10], tol1
    = [0.002,0.005,0.005,0.005,0.01,0.005,0.01,0.005], tol2
    = [0.002,0.005,0.005,0.005,0.01,0.005,0.01,0.005], tol2
    = [0.0001,0.0001,0.0001,0.0001,0.0001,0.0001,0.0001,0.0001], edges=True) :
    = [0.0001,0.0001,0.0001,0.0001,0.0001,0.0001,0.0001,0.0001], edges=True) :
    # INPUT
    # INPUT
    # mesh: trimesh object, the mesh to be analyzed
    # mesh: trimesh object, the mesh to be analyzed
    # pt: (4,2) array, corners of the partitioning grid
    # pt: (4,2) array, corners of the partitioning grid
    # size: float, the mazimum size of a patch
    # size: float, the mazimum size of a patch
    # extra: float, the amount of overlapping for adjacent patches
    # extra: float, the amount of overlapping for adjacent patches
    # degx: (8,) list of int, number of control points along the x axis for the
    # degx: (8,) list of int, number of control points along the x axis for the
    various parts of the blade
    various parts of the blade
    # degy: (8,) list of int, number of control points along the y axis for the
    # degy: (8,) list of int, number of control points along the y axis for the
    various parts of the blade
    various parts of the blade
    # tol1: (8,) list of float, first parameter for the vaious parts of the blade
    # tol1: (8,) list of float, first parameter for the vaious parts of the blade
    # tol2: (8,) list of float, second parameter for the various part of the blade
    # tol2: (8,) list of float, second parameter for the various part of the blade
    # edges: bool, if False does not divide further the edges columns in the grid
    # edges: bool, if False does not divide further the edges columns in the grid
    # OUTPUT
    # OUTPUT
    # damages: (n,3) array, center of the damages detected.
    # damages: (n,3) array, center of the damages detected.
    # values: (n,3) list, values[0]= normalized PC values for the damage, values
    # values: (n,3) list, values[0]= normalized PC values for the damage, values
    [1]=[length, width, depth], values[2]= area
    [1]=[length, width, depth], values[2]= area
    damages = []
    damages = []
    values=[]
    values=[]
    flat, top, bordo1, bordo2, edge1, edge2, edge11, edge22,tempo1, tempo2, tempo3=
    flat, top, bordo1, bordo2, edge1, edge2, edge11, edge22,tempo1, tempo2, tempo3=
    blade_partitioning(mesh, pt, size, extra, edges)
    blade_partitioning(mesh, pt, size, extra, edges)
    if len(flat)>0:#analysis on the airfoil
    if len(flat)>0:#analysis on the airfoil
        straightened, matrixes, centers=allignment(flat, parabole=False)
        straightened, matrixes, centers=allignment(flat, parabole=False)
        danni, error, valori= BSplines_analysis_short(straightened,matrixes,
        danni, error, valori= BSplines_analysis_short(straightened,matrixes,
    centers, 2, 2, degx[0], degy[0], tol1 [0], tol2[0])
    centers, 2, 2, degx[0], degy[0], tol1 [0], tol2[0])
        damages.extend(danni.tolist ())
        damages.extend(danni.tolist ())
        values.extend(valori)
```

        values.extend(valori)
    ```
```

if len (top) $>0$ :\#analysis on the top part
straightened, matrixes, centers=allignment(top, parabole=False)
danni, error, valori= BSplines_analysis_short(straightened,matrixes,
centers, 2, 2, degx[1], degy[1],tol1[1], tol2[1])
damages.extend(danni.tolist())
values.extend(valori)
if len(bordo1)>0:\#analysis on the left border
straightened, matrixes, centers=allignment(bordo1, parabole=False)
danni, error, valori= BSplines_analysis_short(straightened,matrixes,
centers,2,2, degx[2], degy[2],tol1[2],tol2[2])
damages.extend(danni.tolist())
values.extend(valori)
if len(bordo2)>0:\#analysis on the right border
straightened, matrixes, centers=allignment(bordo2, parabole=False)
danni, error, valori= BSplines_analysis_short(straightened, matrixes,
centers,2,2, degx[3], degy[3],tol1[3],tol2[3])
damages.extend(danni.tolist())
values.extend(valori)
if len(edge1)>0:\#analysis on the left edge (first piece)
straightened, matrixes, centers=allignment(edge1, parabole=True)
danni, error, valori= BSplines_analysis_short(straightened,matrixes,
centers,2,2, degx[4], degy[4],tol1[4],tol2[4])
damages.extend(danni.tolist())
values.extend(valori)
if len(edge2)>0:\#analysis on the right edge (first piece)
straightened, matrixes, centers=allignment(edge2, parabole=True)
danni, error, valori= BSplines_analysis_short(straightened, matrixes,
centers,2,2, degx[5], degy[5],tol1[5],tol2[5])
damages.extend(danni.tolist())
values.extend(valori)
if len(edge11)>0:\#analysis on the left edge (second piece)
straightened, matrixes, centers=allignment(edge11, parabole=True)
danni, error, valori= BSplines_analysis_short(straightened, matrixes,
centers,2,2, degx[6], degy[6],tol1[6],tol2[6])
damages.extend(danni.tolist())
values.extend(valori)
if len(edge22)>0:\#analysis on the right edge (second piece)
straightened, matrixes, centers=allignment(edge22, parabole=True)
danni, error, valori= BSplines_analysis_short(straightened,matrixes,
centers,2,2, degx[7], degy[7],tol1[7],tol2[7])
damages.extend(danni.tolist())
values.extend(valori)
damages=np.array (damages)
return damages, values
\#Utility function to save the damages location in a format importable by GOM
def export_gom(found, path, name):
\# INPUT
\# found: (n,3) array, coordinates of the damages locations
\# path: string, directory the file should be saved in
\# name: string, blade name, .stl included
\# OUTPUT
\# a csv file, named ''name', _findings.csv, with the damages location, readable
by GOM
csv=open(path+name[:-3]+, _findings.csv', "w")
for i in range(len(found))
csv.write('Found'+str(i))
for j in found[i]:
number_int=str(int(j))

```
```

8
149
150
151
152
1 5 3

```
```

                number_dec = str(abs(j-int(j)))[2:]
    ```
                number_dec = str(abs(j-int(j)))[2:]
            if number_int='0', and j<0:
            if number_int='0', and j<0:
                number int='-'+number int
                number int='-'+number int
            csv.write('','+number_int+',''+number_dec)
            csv.write('','+number_int+',''+number_dec)
        csv.write('\n')
        csv.write('\n')
    csv.close()
```

    csv.close()
    ```

\section*{A.2.4 analysis_poly}

Subroutines necessary to build the polynomial approximation and perform the analysis.
```

import numpy as np
import scipy as sp
import time
import matplotlib
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import networkx as nx
import trimesh as tm
from utilities import sub_mesh
from utilities import matricexy
from utilities import poly_value
\#Potential damage detector
def parameters_centroids1_short(model, scan, tol):
\# INPUT
\# model: (n,3) array, coordinates of the points of the model with the same
parametrization of the data points.
\# scan: Trimesh object, the surface analized
\# tol: costant value, threshold for differences, called first parameter or
first threshold in the thesis
\# visual: bool, if True show graphic rapresentation of various itermediary
results.
\# OUTPUT
\# damaged_locations: list of trimesh objects, each one represent a dmaged
location
\# root_mean_diff: float, root mean square error between patch and model
diff=model[:, 2] - scan.triangles_center [:, 2]
root_mean_diff=np.sqrt(sum(diff}***2)/len(scan.faces)
diff_filtered=np.zeros(len(diff))
inde\overline{xes = []}
\#\#this does not consider the difference between holes and protrusion.
for n in range(len(diff)):
if abs(diff[n])>tol:
indexes.append(n)
diff_filtered[n]=np.sign(diff[n])*(abs(diff[n])-tol)\#I'm not sure the
sign is necessary
if indexes==[]:
return indexes , root_mean_diff
diff_pt=scan.triangles_ce\overline{n}ter [:, : 2][indexes]
indexes=np.array(indexes)
\#we create a list of points with damages and a figure to show them
diff_pt_value=np.vstack((diff_pt.T, diff_filtered[indexes])).T
mesh1=scan . copy ()
mesh1.update_faces(indexes)

```
```

graph = nx.Graph()
graph.add_nodes_from(np.arange(len(scan.faces)))
graph.add_edges_from(mesh1.face_adjacency)
groups = \overline{nx.connected_components(graph)\#}
gruppi=np.array(list(groups))
l=[]
singoletti=[]
usati=0
n=0
while n<len(gruppi) and usati<len(indexes):
i=gruppi[n]
n+=1
usati+=len(list(i))
if len(i)>1:
l.append(indexes[list(i)])
else:
singoletti.extend(indexes[list (i)])
damaged_locations=[]
graph = ñ..Graph()
graph.add_nodes_from(np.arange(len(scan.faces)))
graph.add_edges_from(scan.face_adjacency)
for j in l
indici=[]
for i in j:
ll=list(graph[i])
indici.extend(ll)
j=np.array(indici)
indici=[]
for i in j:
ll=list(graph[i])
indici.extend(ll)
indici=np.array(indici)
j=np.unique(indici)
new_mesh=sub_mesh(scan,j)
damaged_locations.append(new_mesh)
for j in singoletti:
j=graph.neighbors(j)
if len(j)>0:
for k in range(5):
indici=[]
for i in j:
ll=list(graph[i])
indici.extend(ll)
indici=np.array(indici)
j=np.unique(np.append(indici , j))
new_mesh=sub_mesh(scan,j)
damaged_locations.append(new_mesh)
return damaged_locations , root_mean_diff
\#PCA evaluator
def parameters_centroids2_short(model,scan) :
\# INPUT
\# model: (n,3) array, coordinates of the points of the model with the same
parametrization of the data points. Damage only.
\# scan: Trimesh object, the surface analized. Damage only.
\# OUTPUT
\# S: (3,) array, PC values from the covariance matrix on 3D points
\# V: (3,3) matrix, PC directions from the covariance matrix on 3D points

```
```

    #re-evaluate the differences
    diff=model[:, 2] - scan.triangles_center[:, 2]
    #if less than 2 point are present on the damage no meaningful analysis can be
    performed. the process is then interrupted and returns only zeros.
    if len(diff)<2:
        print("The damage is too small here")
        a=np.array ([0, 0,0])
        b=np.array ([0,0,0],[0,0,0],[0,0,0])
    return a,b
    #build a new set of point with the same xy coordinates and the differences as
    z values
    diff pt=scan.triangles center [:,:2]
    diff_pt_value=np.vstack
    #build the covariance matrix
    mean=np.mean(diff_pt_value, axis=0)
    covariance=np.dot((diff_pt_value-mean).T, diff_pt_value-mean)
    #Use the SVD decomposition to compute the PC values and vectors
    U,S,V=np.linalg.svd(covariance)
    #print(S1/np.linalg.norm(S1))
    #Return the principal values and vectors for covariance and correlation
    matrixes
    return S, V
    \#Patchwise polynomial analysis
def polynomial_analysis_short(dritte, matrici, centri, degx, degy,tol1=0.005,tol2
=0.0001, shape=True):

# INPUT

# dritte: (n,) array of trimesh objects, each mesh is centered in the origin and

        oriented as described in the thesis.
    
# matrici: (n,) array of (3,3) matrixes, each one is the rotation necessary to

    revert the respective mesh to its original orientation
    
# centri: (n,) array of (3,) vectors, the translation necessary to move the

    respective mesh to its original position
    # degx: int, degree of the polinomial along the x axis
    
# degy: int,degree of the polinomial along the y axis

# tol1: float, first threshold or parameter, as described in the thesis

# tol2: float, second threshold or parameter, as described in the thesis

# shape: bool, if True an additional filter is addet to the results

# OUTPUT

# danni: (m,3) array, coordinates of the centers of the damages detected

# error: (n,2) array, contains the RMSE and the number of faces for each patch

# valori: (m,3) array, valori[0]= normalized PC values for the damages, valori

    [1]=[length, width, depth], valori[2]=area
    numero=1
danni=[]
valori=[]
faces_number = []
mean_error =[]
for m in dritte:
matrice=matrici[numero-1]
centro=centri[numero-1]
faces_number.append(len(m.faces))
print("\#||||||||||||||||||\#")
print('Mesh number %d' %numero)
print("\#\#|||||||||||||||||||')
print("Centered in ",centro)
numero+=1
54 \#||||||||||||||||||||||

```
\#Build the matrix required to perform the least square approximation. Uses a 2 D polynomial of degree degx*degy.
\#build the matrix with \(\left[1, x, x^{\wedge} 2, \ldots, x^{\wedge} \operatorname{deg} x, y * 1, y * x, \ldots, y^{\wedge} \operatorname{deg} y * x^{\wedge} \operatorname{degx}\right]\) as row for each face center in the mesh
\(A=\) matricexy (m.triangles_center [:, : 2] , degx, degy)
\#Perform the least square approximation to determine the coefficient of the polynomial
\(x=n p . l i n a l g . l s t s q\left(A, n p . a s a r r a y\left(m . t r i a n g l e s \_c e n t e r[:, ~ 2]\right) . f l a t t e n()\right) ~\)
\#inside is a \(n\) array of the \(z\)-values of the polynomial found on the
coordinates of the faces center
inside=poly_value (m. triangles_center [:, : 2] , x[0], degx, degy, A)
\#poly points are the 3 D version of inside, with x and y coordinates too
poly_points=np.vstack ((m. triangles_center[:, : 2].T, inside)).T
\#we apply parameter_centroids1_short
lista, medio=parameters_centroids1_short (poly_points,m,tol1)
\#we add the new root mean square ēror to the list
mean_error. append (medio)
\#if there are damages to analize
damaged_zones \(=[]\)
\#each sub-mesh extracted is then analyzed independly to determine it is an impact damage or not.
if len (lista) \(>0\) :
for \(i\) in lista:
\#the same as poly_points but only for the faces centers of the damaged
location
spl_pts=np.asarray ([i.triangles_center[:, 0], i.triangles_center [:, 1] ,
poly_value(i.triangles_center [:, : 2] , x[0], degx, degy)]).T
\#we apply parameters_centroids2_short
S_cov, V_cov=parameters_centroids2_short(spl_pts,i)
\#we normalize the vector of the \(\overline{P C}^{-}\)values, its the real parameter to indicate if an imperfection is a damage or not.
parametro=S_cov/np.linalg.norm (S_cov)
\# find the center of the damage
mean \(=\) np. mean (i.triangles_center, axis \(=0\) )
\#and roto-translate it back to the original position
original \(=\) np. dot (matrice, mean \()[0]+\) centro [0]
\#print some useful informations
print(" ")
print("Center of the damage=", original)
print("Un-normalized parameter=", S_cov)
print("Considered parameters= " , parametro)
print ("Linearity param=", (S_cov[0]-S_cov[1])/S_cov[0])
print("Planarity param=", (S_cov[1]-S_cov[2])/S_cov[0])
print("Sfericity param=", S_cov[2]/S_cov[0])
print ("Anisotropy=", (S_cov[0]-S_cov[2])/S_cov[0])
print ("Eighenentropy=",-S_cov[0]*np. \(\log \left(S_{-}^{-} \operatorname{cov}[0]\right)-S \_c o v[1] * n p . \log \left(S \_c o v\right.\)
[1])-S_cov[2]*np. log (S_cov[2]))
print ("Curvature change=", S_cov[2]/(S_cov[0]+S_cov[1]+S_cov[2]))
centri_i=i.vertices \(-n \mathrm{p}\). mean \(\overline{(i}\). vertices, axis \(=0)\)
ruotati=np. asarray (np. dot (V_cov, centri_i.T) )
\#compute the damage dimensions
lunghezza \(=\max (\) ruotati \([0,:])-\min (r u o t a t i[0,:]) \#\) lenght
larghezza=max (ruotati[1, :])-min(ruotati \([1,:])\) \#width
profondita \(=\max (\) ruotati[2,: \(])-\min (\) ruotati \([2,:])\) \#depth
print('Max distances=',[lunghezza, larghezza, profondita])
\#evaluate the area and volume of the damage as the one of the convex hull containing it
\#ruotati=ruotati.T
\#hull2D=sp.spatial. ConvexHull(ruotati[:, 2]) \#using only the first two coordinates we obtain the area
```

    #print("Area of the triangles in the convex hull =",hull2D.volume)
    ```
    #print("Area of the triangles in the convex hull =",hull2D.volume)
        #hull3D=sp.spatial.ConvexHull(ruotati[:,:])#using all the coordinates
        #hull3D=sp.spatial.ConvexHull(ruotati[:,:])#using all the coordinates
    allow the volume to be extracted
    allow the volume to be extracted
        #print("Volume of the 3D convex hull =",hull3D.volume)
        #print("Volume of the 3D convex hull =",hull3D.volume)
        #translate and orientate the damage so that the incavation/bump is as
        #translate and orientate the damage so that the incavation/bump is as
    vertical as possible, to determine its dimensions, then printed
    vertical as possible, to determine its dimensions, then printed
        centri_i=i.triangles_center
        centri_i=i.triangles_center
        ruotati=np.asarray(np.dot(V_cov, centri_i.T))
        ruotati=np.asarray(np.dot(V_cov, centri_i.T))
        lunghezza=max(ruotati[0,:])-min(ruotati[0,:])#length
        lunghezza=max(ruotati[0,:])-min(ruotati[0,:])#length
        larghezza=max(ruotati[1,:])-min(ruotati[1,:])#width
        larghezza=max(ruotati[1,:])-min(ruotati[1,:])#width
        profondita=max(ruotati[2,:])-min(ruotati[2,:])#depth
        profondita=max(ruotati[2,:])-min(ruotati[2,:])#depth
        print('length=',lunghezza,' width=',larghezza,' depth=',profondita)
        print('length=',lunghezza,' width=',larghezza,' depth=',profondita)
        ruotati=ruotati.T
        ruotati=ruotati.T
        hull2D=sp.spatial.ConvexHull(ruotati[:,:2])#using only the first two
        hull2D=sp.spatial.ConvexHull(ruotati[:,:2])#using only the first two
    coordinates we obtain the area
    coordinates we obtain the area
        area=hull2D.volume
        area=hull2D.volume
        print(" ")
        print(" ")
        #if the 3rd normalized PC value is higher than tol2 the location is an
        #if the 3rd normalized PC value is higher than tol2 the location is an
    impact damage
    impact damage
        if parametro[2]>=tol2:
        if parametro[2]>=tol2:
            if shape:#if shape=-True the ratio between the firs and second dimension
            if shape:#if shape=-True the ratio between the firs and second dimension
        is also cheched. If the 'length' is more than 3 times the 'width' is too
        is also cheched. If the 'length' is more than 3 times the 'width' is too
    scratch-like and is discarded. Working with parametro only gives an indication
    scratch-like and is discarded. Working with parametro only gives an indication
    of the damage size, an additional filter based on the real dimentions can be
    of the damage size, an additional filter based on the real dimentions can be
    added.
    added.
            if parametro[1]>=(parametro[0]/3):#acceptable shape
            if parametro[1]>=(parametro[0]/3):#acceptable shape
                        print("#||||||||||||||||||||||||||||||||||||||||||||||Is acceptable
                        print("#||||||||||||||||||||||||||||||||||||||||||||||Is acceptable
    ##|||||||||||||||||||||||||||||||||||||||||||||"')
    ##|||||||||||||||||||||||||||||||||||||||||||||"')
        print(" ")
        print(" ")
        valori.append([S_cov/np.linalg.norm(S_cov),[lunghezza,larghezza,
        valori.append([S_cov/np.linalg.norm(S_cov),[lunghezza,larghezza,
    profondital, areal)
    profondital, areal)
        danni.append(original)
        danni.append(original)
            else:
            else:
                print('Too scratch-like')#in this case is discarded
                print('Too scratch-like')#in this case is discarded
            else:
            else:
        print("#|||||||||||||||||||||||||||||||||||||||||||||||Is acceptable
        print("#|||||||||||||||||||||||||||||||||||||||||||||||Is acceptable
    ##||||||||||||||||||||||||||||||||||||||||||||||#')
    ##||||||||||||||||||||||||||||||||||||||||||||||#')
        print(" ")
        print(" ")
                valori.append([S_cov/np.linalg.norm(S_cov),[lunghezza,larghezza,
                valori.append([S_cov/np.linalg.norm(S_cov),[lunghezza,larghezza,
    profondita], area])
    profondita], area])
        danni.append(original)
        danni.append(original)
        #otherwise is a faux positive
        #otherwise is a faux positive
        else:
        else:
            print("Is not acceptable")
            print("Is not acceptable")
        print("##")
        print("##")
danni=np.asarray(danni)
danni=np.asarray(danni)
#if err_visual is True it shows a graph of the root mean square error
#if err_visual is True it shows a graph of the root mean square error
error=[mean_error, faces_number]
error=[mean_error, faces_number]
print("Total root mean square error=",np.sqrt(np.dot(np.asarray(error[0]) **2,np.
print("Total root mean square error=",np.sqrt(np.dot(np.asarray(error[0]) **2,np.
    asarray(error[1]))/sum(error[1])))
    asarray(error[1]))/sum(error[1])))
return danni, error, valori
```

return danni, error, valori

```

\section*{A. 3 Partition}

The following function are used to extract the patches to be analyzed from the scan surface.

\section*{A.3.1 partition_main}

Summary script to perform the partitioning of the surface.
```

import numpy as np
import scipy as sp
import time
import matplotlib
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
plt.rcParams["figure.figsize"] = [10,10]
import networkx as nx
import trimesh as tm
from utilities import cut
from utilities import sub_mesh
from utilities import workbench
from partition_grid_cut import grid_maker
from partition_grid_cut import windowing
from partition_refinement import border_extra
\#This function refine all the sub meshes extracted from windowing. this is the
function to be modified in case blades with different shapes are tested. As an
example, if the blade is attached to a structure also on the top the main
mart should encompass all the cells.
def partitioning(sub_grid,sub_vertices, visual=False):
\# INPUT
\# sub_grid: 2D array of trimesh object, to understand its structure see
windowing
\# sub_vertices: 2D array of 2D points, the corners of the grid cells, for its
structure see windowing
\# visual: bool, if True the middle steps are shown, slowing the process
considerably
\# OUTPUT
\# flat: list, patches from the airfoil
\# top: list, patches from the top row
\# bordo1: list, patches from the left edge border
\# bordo2: list, patches from the right side border
\# edge1: list, first patches from the left side edge
\# edge2: list, first patches from the right side edge
\# edge11: list, second patches from the left side edge
\# edge22: list, second patches from the right side edge
\# tempo: float, total time necessary for this function (seconds)
t 5=time.time()
flat=[]
bordo1=[]
bordo2=[]
top = []
edge1 = []
edge11=[]
edge2=[]
edge22=[]
\#The starting points for the edges, corners and top partitioning are chosen to
be on a part as far away as possible from the edges.
\#\#\#||||MAIN PART
for i in range(len(sub_grid)-1):\#analyze all the cells exept for the upper row
That part is analyzed separately because of its features
for j in range(len(sub_grid[i])):
if j==0:
\#Meshes on the left edge, border extra is applied
normal_pt=(sub_vertices[i,j][2]+sub_vertices[i,j][3])/2

```
```

    lati , bordo, edge=border_extra(sub_grid[i,j], normal_pt,tol=0.01,
    ```
    lati , bordo, edge=border_extra(sub_grid[i,j], normal_pt,tol=0.01,
    visual=visual)
    visual=visual)
        for k in lati:
        for k in lati:
            bordo1.append(k)
            bordo1.append(k)
        for k in bordo:
        for k in bordo:
            edge1.append(k)
            edge1.append(k)
        for k in edge:
        for k in edge:
            edge11.append(k)
            edge11.append(k)
        elif j==(len(sub_grid[i]) -1):
        elif j==(len(sub_grid[i]) -1):
        #Meshes on the right edge, border extra is applied
        #Meshes on the right edge, border extra is applied
        normal_pt=(sub_vertices[i,j][0]+ sub_vertices[i,j][1])/2
        normal_pt=(sub_vertices[i,j][0]+ sub_vertices[i,j][1])/2
        lati, bordo, edge=border_extra(sub_grid[i,j], normal_pt,tol=0.01,
        lati, bordo, edge=border_extra(sub_grid[i,j], normal_pt,tol=0.01,
    visual=visual)
    visual=visual)
        for k in lati:
        for k in lati:
            bordo2.append(k)
            bordo2.append(k)
        for k in bordo:
        for k in bordo:
            edge2.append(k)
            edge2.append(k)
        for k in edge:
        for k in edge:
            edge22.append(k)
            edge22.append(k)
        else:#Airfoil patches
        else:#Airfoil patches
        #Those portions sides are easily distinguishable with a
        #Those portions sides are easily distinguishable with a
    connectivity graph
    connectivity graph
        graph = nx.Graph()
        graph = nx.Graph()
        graph.add_edges_from(sub_grid[i, j].face__adjacency)
        graph.add_edges_from(sub_grid[i, j].face__adjacency)
        groups = nx.connected_components(graph)
        groups = nx.connected_components(graph)
        lista=list(groups)
        lista=list(groups)
        if len(lista)==1:
        if len(lista)==1:
            print('EDGE sbaglaito')
            print('EDGE sbaglaito')
            print('Mesh coord in the grid:(',i,',',j,'')')
            print('Mesh coord in the grid:(',i,',',j,'')')
        if len(lista)}>2:# if some extra piece is included, it notifies th
        if len(lista)}>2:# if some extra piece is included, it notifies th
    error
    error
            print('Mesh coord in the grid:(',i,',',j,')')
            print('Mesh coord in the grid:(',i,',',j,')')
            print('Position in the flat parts:', len(flat))
            print('Position in the flat parts:', len(flat))
            print('Corrisponding number of points:')
            print('Corrisponding number of points:')
            for k in lista:
            for k in lista:
                print(len(k))
                print(len(k))
                lato=list(k)
                lato=list(k)
                    if len(lato)>10:
                    if len(lato)>10:
                                    temp_mesh=sub_mesh(sub_grid[i,j], lato)
                                    temp_mesh=sub_mesh(sub_grid[i,j], lato)
                                    flat.append(temp_mesh)
                                    flat.append(temp_mesh)
                lato1=list(lista[0])
                lato1=list(lista[0])
                temp_mesh=sub_mesh(sub_grid[i,j], lato1)
                temp_mesh=sub_mesh(sub_grid[i,j], lato1)
                flat.append(temp_mesh)
                flat.append(temp_mesh)
                lato2=list(lista[1])
                lato2=list(lista[1])
                temp_mesh=sub_mesh(sub_grid[i,j], lato2)
                temp_mesh=sub_mesh(sub_grid[i,j], lato2)
                        flat.append(temp_mesh)
                        flat.append(temp_mesh)
#H||END MAIN PART
#H||END MAIN PART
##|#TOP ROW
##|#TOP ROW
#The top sstrip is more complex because of the corners and the flat edge on
#The top sstrip is more complex because of the corners and the flat edge on
top, is considered separately
top, is considered separately
    for j in range(len(sub_grid[-1])):
    for j in range(len(sub_grid[-1])):
        if j==0:
        if j==0:
        #Left corner, we ignore the edges, but the borders are saved as top
        #Left corner, we ignore the edges, but the borders are saved as top
        normal_pt=sub_vertices[-1,j][3]
        normal_pt=sub_vertices[-1,j][3]
        lati , bordo, edge=border_extra(sub_grid[-1,j], normal_pt)
        lati , bordo, edge=border_extra(sub_grid[-1,j], normal_pt)
        for k in lati:
```

        for k in lati:
    ```
```

        top.append(k)
        elif j==(len(sub_grid[-1]) -1):
        #Right corner, we ignore the edges, but the borders are saved as top
        normal_pt=sub_vertices[ - 1, j][0]
        lati, bordo, edge=border_extra(sub_grid[-1,j], normal_pt)
        for k in lati:
            top.append(k)
        else:
        #Portion on the top corners excluded
        normal_pt=(sub_vertices[ - 1, j][0]+ sub_vertices[ - 1,j][3])/2
        lati, bordo, edge=border_extra(sub_grid[-1,j], normal_pt, tol=0.005)
        if len(lati)>2:
            print('Mesh coord in the grid:(',i,',',j,')')
            print('Position in the flat parts:', len(flat))
            print('Corrisponding number of points:')
            for k in lati:
                print(len(k.faces))
        for k in lati:
            top.append(k)
    t6=time.time()
    tempo=(t6-t5)
    print('Time necessary to divide the two sides and the edges from each other=',
    tempo)
    return flat, top, bordo1, bordo2, edge1, edge2, edge11, edge22,tempo
    def blade_partitioning(mesh, pt, size=5., extra=1., edges=True):
\# INPUT
\# mesh: trimesh object, the whole blade scan
\# pt: (4,2) array, corners of the partitioning grid
\# size: float, maximum dimansion for a cell, mm,
\# extra: float, expansion of the cells to allow overlapping
\# edges: bool, if False does not divide further the edges columns in the grid
\# OUTPUT
\# flat: list, patches from the airfoil
\# top: list, patches from the top row
\# bordo1: list, patches from the left edge border
\# bordo2: list, patches from the right side border
\# edge1: list, first patches from the left side edge
\# edge2: list, first patches from the right side edge
\# edge11: list, second patches from the left side edge
\# edge22: list, second patches from the right side edge
\# tempo1: float, time necessary to remove the base
\# tempo2: float, time necessary to cut along the grid
\# tempo3: float, time necessary to refine the cells.
t 3=time.time()
lab_rat=cut(mesh,pt)
t4=-time.time()
print('Mesh imported and selection of the blade portion completed in ',t4-t3,
seconds.')
sub_grid, sub_vertices, tempo2=windowing(lab_rat, grid, extra)
plt.plot(grid[:,:,0],grid[:,:,1], c='r')
for i in grid:
plt.plot(i[:,0],i[:, 1], c='r')
mesh_pt=mesh.triangles_center
plt.s.satter(mesh_pt[range(0, len(mesh_pt),100),0], mesh_pt[range(0, len(mesh_pt)
,100), 2], c='c', marker='.', s=1)
plt.title('How the portioning is done')
plt.show()

```
```

162
163

```
print('Grid partitioning completed in ',tempo2,' seconds.')
```

print('Grid partitioning completed in ',tempo2,' seconds.')
flat, top, bordo1, bordo2, edge1, edge2, edge11, edge22, tempo3=partitioning(sub_grid,
flat, top, bordo1, bordo2, edge1, edge2, edge11, edge22, tempo3=partitioning(sub_grid,
sub_vertices)
sub_vertices)
print('Mesh refining completed in ',tempo3,' seconds.')
print('Mesh refining completed in ',tempo3,' seconds.')
return flat,top,bordo1,bordo2, edge1, edge2, edge11, edge22,(t4-t 3),tempo2,tempo3
return flat,top,bordo1,bordo2, edge1, edge2, edge11, edge22,(t4-t 3),tempo2,tempo3
def blisk_partitioning(mesh, pt=np.array ([[x, x],[x, x],[x, x],[x, x]]),
def blisk_partitioning(mesh, pt=np.array ([[x, x],[x, x],[x, x],[x, x]]),
size=2., extra=0.5, pt1=np.array ([[x, x],[x, x],[x, x],[x,x]]),
size=2., extra=0.5, pt1=np.array ([[x, x],[x, x],[x, x],[x,x]]),
pt2=np.array ([[x, x],[x, x],[x, x],[x, x]]), edges=True):
pt2=np.array ([[x, x],[x, x],[x, x],[x, x]]), edges=True):
\# INPUT
\# INPUT
\# mesh: trimesh object, the whole blade scan
\# mesh: trimesh object, the whole blade scan
\# pt: (4,2) array, corners of the partitioning grid
\# pt: (4,2) array, corners of the partitioning grid
\# pt1: (4,2) array, corners of the partitioning grid for the lower part
\# pt1: (4,2) array, corners of the partitioning grid for the lower part
\# pt2: (4,2) array, corners of the partitioning grid for the upper part
\# pt2: (4,2) array, corners of the partitioning grid for the upper part
\# size: float, maximum dimansion for a cell, mm
\# size: float, maximum dimansion for a cell, mm
\# extra: float, expansion of the cells to allow overlapping, mm
\# extra: float, expansion of the cells to allow overlapping, mm
\# edges: bool, if False does not divide further the edges columns in the grid
\# edges: bool, if False does not divide further the edges columns in the grid
\# OUTPUT
\# OUTPUT
\# flat: list, patches from the airfoil
\# flat: list, patches from the airfoil
\# top: list, patches from the top row
\# top: list, patches from the top row
\# bordo1: list, patches from the left edge border
\# bordo1: list, patches from the left edge border
\# bordo2: list, patches from the right side border
\# bordo2: list, patches from the right side border
\# edge1: list, first patches from the left side edge
\# edge1: list, first patches from the left side edge
\# edge2: list, first patches from the right side edge
\# edge2: list, first patches from the right side edge
\# edge11: list, second patches from the left side edge
\# edge11: list, second patches from the left side edge
\# edge22: list, second patches from the right side edge
\# edge22: list, second patches from the right side edge
\# tempol: float, time necessary to remove the base
\# tempol: float, time necessary to remove the base
\# tempo2: float, time necessary to cut along the grid
\# tempo2: float, time necessary to cut along the grid
\# tempo3: float, time necessary to refine the cells.
\# tempo3: float, time necessary to refine the cells.
t 3=time.time()
t 3=time.time()
\#extra_border=np. array ([[- extra, - extra],[- extra, extra],[ extra, extra],[ extra,-
\#extra_border=np. array ([[- extra, - extra],[- extra, extra],[ extra, extra],[ extra,-
extra]])
extra]])
lab_rat=cut(mesh,pt)\#+extra_border)
lab_rat=cut(mesh,pt)\#+extra_border)
grid1=grid_maker(pt1, size, edges)
grid1=grid_maker(pt1, size, edges)
plt.plot(grid1[:,:,0],grid1[:,:,1], c='r'')
plt.plot(grid1[:,:,0],grid1[:,:,1], c='r'')
for i in grid1:
for i in grid1:
plt.plot(i[:,0],i[:, 1], c='r')
plt.plot(i[:,0],i[:, 1], c='r')
grid1=grid1[:-1]
grid1=grid1[:-1]
grid2=grid_maker(pt2,size, edges,forced_width=grid1.shape[1] - 2)
grid2=grid_maker(pt2,size, edges,forced_width=grid1.shape[1] - 2)
plt.plot(grid2[:,:,0],grid2[:,:, 1], c='r')
plt.plot(grid2[:,:,0],grid2[:,:, 1], c='r')
for i in grid2:
for i in grid2:
plt.plot(i[:,0], i[:, 1], c='r')
plt.plot(i[:,0], i[:, 1], c='r')
grid=np.vstack((grid1,grid2))
grid=np.vstack((grid1,grid2))
t4=time.time()
t4=time.time()
print('Mesh imported and selection of the blade portion completed in ',t4-t3,
print('Mesh imported and selection of the blade portion completed in ',t4-t3,
' seconds.')
' seconds.')
sub_grid,sub_vertices, tempo2=windowing(lab_rat,grid, extra)
sub_grid,sub_vertices, tempo2=windowing(lab_rat,grid, extra)
mesh_pt=mesh.triangles_center
mesh_pt=mesh.triangles_center
plt.scatter(mesh_pt[range(0, len(mesh_pt),100),0],mesh_pt[range(0, len(mesh_pt)
plt.scatter(mesh_pt[range(0, len(mesh_pt),100),0],mesh_pt[range(0, len(mesh_pt)
,100),2],c='c', marker='.', s=1)
,100),2],c='c', marker='.', s=1)
plt.title('How the portioning is done')
plt.title('How the portioning is done')
plt.show()

```
plt.show()
```

```
print('Grid partitioning completed in ',tempo2,' seconds.')
    flat, top, bordo1, bordo2, edge1, edge2, edge11, edge22,tempo3=partitioning(sub_grid,
    sub_vertices)
    print('Mesh refining completed in ',tempo3,' seconds.')
    return flat, top,bordo1, bordo2, edge1, edge2, edge11, edge22,(t4-t3), tempo2, tempo3
def allignment(meshes, parabole=False):
    # INPUT
    # meshes: list of trimesh objects, the meshes to be re-oriented
    # parabole: bool, if true a second rotation is applied to the (y,z) coord
    # OUTPUT
    # straightened: list of trimesh objects, the re-oriented meshes
    # matrici: list of (3,3) matrixes, the rotation matrixes necessary to put the
    mesh in its original position
    # centers: list of (3,) array, the previous centers of the meshes
    straightened=[]
    matrixes=[]
    centers=[]
    if parabole:
        for i in meshes:
            #the mesh is alligned with the PC direction and translated so that its
        center is on the origin
            rotation0, center0, new_coord0=workbench(i)
            mm=tm.base.Trimesh(new_coord0,i.faces)
            #a table to compare the errors (and the parameters used to evaluate it
    ) is initialized
            err=np.vstack((np.linspace(-np.pi/6,np.pi/6,20),np.zeros((4, 20))))
            punti=mm. vertices
            n=0
            #For the angles chosen (20 equispaced between [ - 30,30]) the mesh is
    rotated around the x axis. the error obtained with a parabola approximant of
    the point projected along the x axis is evaluated, and used to determine the
    best angle for the adjustment.
    for theta in np.linspace(-np.pi/6,np.pi/6,20):
            #rotation matrix
            M=np.array([[np.cos(theta), -np.sin(theta)],[np.sin(theta), np.cos(
theta) ]])
            #rotated points
            pt=np.array(np.dot(M, punti[:, 1:].T).T)
            #matrix to evaluate the parabola values
            A=np.vstack((np.ones((1,len(punti))),pt[:,0].T,pt[:,0]**2)).T
            #best parabola approximation and relative error
            parametri,e,_,_=np.linalg.lstsq(A, pt[:, 1])
            err [1,n]=e
            a=parametri[2]
            err[2,n]=a
            b=parametri[1]
            err[3,n]=b
            c=parametri[0]
            err [4,n]=c
            n+=1
            #find the angle with minimal error
            indice=np.argmin(err [1,:])
            theta=err[0,indice]
            a=err[2,indice]
            b=err[3, indice]
            c=err[4,indice]
            #apply the rotation
            M=np.array([[np.cos(theta), -np.sin(theta)],[np.sin(theta),np.cos(theta
)]!)
```

```
6 9
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
```

```
    pt=np.array(np.dot(M, punti[:, 1:].T).T)
```

    pt=np.array(np.dot(M, punti[:, 1:].T).T)
    punti=np.vstack((punti[:,0],pt.T)).T
    punti=np.vstack((punti[:,0],pt.T)).T
    mm=tm.base.Trimesh(punti,mm. faces)
    mm=tm.base.Trimesh(punti,mm. faces)
    ##||||||||||||||||||||||#
    ##||||||||||||||||||||||#
    straightened.append (mm)
    straightened.append (mm)
    #save teh product of the original rotation and the second one.
    #save teh product of the original rotation and the second one.
    matrixes.append ([np.dot(np.linalg.inv(rotation0),np.vstack(([1,0,0],np
    matrixes.append ([np.dot(np.linalg.inv(rotation0),np.vstack(([1,0,0],np
    hstack(([[0],[0]],M)))))])
    hstack(([[0],[0]],M)))))])
    centers.append(center0)
    centers.append(center0)
    else:
else:
for i in meshes:\#the same without the second rotation
for i in meshes:\#the same without the second rotation
rotation0 , center0, new_coord0=workbench(i)
rotation0 , center0, new_coord0=workbench(i)
mm=tm.base. Trimesh(new_coord0,i.faces)
mm=tm.base. Trimesh(new_coord0,i.faces)
straightened.append (mm)
straightened.append (mm)
matrixes.append([np.linalg.inv(rotation0)])
matrixes.append([np.linalg.inv(rotation0)])
centers.append(center0)
centers.append(center0)
return straightened, matrixes, centers

```
    return straightened, matrixes, centers
```


## A.3.2 partition refinement

Functions used to divide the window extracted through the grid and re-orientate the patches obtained.

```
import numpy as np
import scipy as sp
import time
import matplotlib
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import networkx as nx
import trimesh as tm
from utilities import sub_mesh
from utilities import matricexy
from utilities import poly_value
#The following functions are necessary to apply the second partitioning method
    discussed in the thesis.
#patch_refiner is the function used to separate the border portions from the edge
    ones
def patch_refiner(mesh, face_list, scalare, tol, degree= = , visual=False):
    # INPUT
    # mesh: trimesh object, mesh in one of the edge, corner or top cells
    # face_list: (n,) array, indexes of the faces on one side of the mesh
    # scalare: (n,) array, scalar product values of the mesh face normals with a
    reference direction
    # tol: float, maximum distance between data and approximant to be added to the
        building set
    # degree: int, degree of the polynomial approximant
    # visual: bool, if True additional figures are shown
    # OUTPUT
    # face_list1: (m,) array, indexes of the faces in the border piece.
    #The first thing to do is to select the first building set. We select the 20%
    of the faces with normal more paralled to the referece direction e.g. the one
    with higher scalar product.
    nucleo=[]
    #this is the value of the 75% higher scalr product value
    valore=np.sort(scalare[face_list])[int(len(face_list)*0.75)]
    nucleo=[]
    for i in range(len(face_list)):
```

```
    if scalare[face_list[i]]>valore:
        nucleo.append(i)
nucleo=np.array(nucleo)
a=face_list[nucleo]#building set
vertici=mesh.triangles_center[a]#those are the centers of the selected faces.
To simplify the approximant construction the mesh is then oriented using the
PCA results over the selected points, as in 'workbench'
    #find the mean point, or center of mass
    center=np.mean(vertici, axis=0)
    #translate the points
    translated=vertici-center
    #build the covariance matrix
    covariance_matrix=np.dot(translated.T, translated)
    #find the eighenvectors
    U, s,V=np.linalg.svd(covariance_matrix)
    v1=V[0]
    v2=V[1]
    #v3=V[2] #just to be sure we find the last vector with a cross product to
    avoid signs problems
    plane_normal=np.cross(v1,v2)
    #and build the rotation matrix
    rotation=np.asanyarray([v1, v2, plane_normal])
    rotation=np.matrix(rotation)
    #we define the new coordinates
    translated=mesh.triangles_center[face_list]-center
    pt=np.asarray(np.dot(rotation, translated.T).T)
    #the border definition process is repeated up to 10 times
    for n in range(10):
    ####the mesh is oriented to better approximate the building set
    a=face_list[nucleo]
    vertici=mesh.triangles_center[a]
    #we find the mean point, or center of mass
    center=np.mean(vertici, axis=0)
    #and translate the points
    translated=vertici-center
    #we build the covariance matrix
    covariance_matrix=np.dot(translated.T, translated)
    #we find the eighenvectors
    U, s,V=np.linalg.svd(covariance_matrix)
    v1=V[0]
    v2=V[1]
    #v3=V[2] #just to be sure we find the last vector with a cross product to
    avoid signs problems
    plane_normal=np.cross(v1,v2)
    #and build the rotation matrix
    rotation=np.asanyarray([v1,v2, plane_normal])
    rotation=np.matrix(rotation)
    #we define the new coordinates
    translated=mesh.triangles_center[face_list]-center
    pt=np.asarray(np.dot(rotation, translated.T).T)
    ###
    #a polynomial approximant is built from the (surprise surprise) building
    set and used to define the next one
    A=matricexy(pt[nucleo,:2], degree,degree)
    #Perform the least square approximation to determine the coefficient of
    the polynomial
    x=np.linalg.lstsq(A, np.asarray(pt[nucleo, 2]).flatten())
    #approx is a collection of the values of the polynomial found on the
coordinates of the faces center
    approx=poly_value(pt[:,:2],x[0], degree,degree)
```

```
*
86
87
88
89
90
91
92
93
    desctibed in the thesis
def border_extra(mesh,pt,tol=0.005,visual=False):
    # INPUT
    # mesh: trimesh object, mesh in one of the edge, corner or top cells
    # pt: (2,) array, point used to determine the reference direction
    # tol: float, maximum distance between data and approximant to be added to the
    building set (for borders)
```

```
# visual: bool, if True additional figures are shown
# OUTPUT
# lati: list, mesh of the border parts
# bordi: list, first mesh of the edge parts
# spigolo: list, second mesh of the edge parts
#We chose the triangular face closest to pt as our reference
faccia=np.argmin(np.linalg.norm(mesh.triangles_center[:, 0:3:2] - pt,axis=1))
#We consider a wider zone to get a better result. This piece of code find all
the neighbor faces adjacent to the selected one
####
graph = nx.Graph()#
graph.add_edges_from(mesh.face_adjacency)#
indici=[]
l=list(graph.neighbors(faccia))
indici.extend(l)
indexes=np.sort(np.unique(indici))
####
#If we want to use a larger number of faces to build the normal we have to
repeat this part
for i in indexes:
    l=list(graph.neighbors(i))
    indici.extend(l)
indici=np.array(indici)
indici=np.append(indici, indexes)
indexes=np.sort(np.unique(indici))
#||#
#The reference normal will be the mean value of the one of the selected faces
normale=np.mean(mesh.face_normals[indexes], axis=0)
#And we normalize it.
normale=normale/np.linalg.norm(normale)
#We compute the scalar product between each face normal and 'normale'
scalare=np.dot(mesh.face_normals, normale)
lati=[]
bordi=[]
spigolo=[]
face_list1=[]
face_list2=[]
#the mesh is roughly divided in two overlapping parts that must completely
contain the two border pieces. The values of 'scalare' are used to select the
faces.
valore1=np.sort(scalare)[int(len(scalare)/3)] #the 70% of faces with the
higher scalar values are in the first part
valore 2=np.sort(scalare)[2*int(len(scalare)/3)] #the 70% of faces with the
lower scalar values are in the second part
for i in range(len(mesh.faces)):
    #If the scalar product is higher than valore1 the face is on the first
side.
    if scalare[i]>=valore1:
                face_list1.append(i)
    #If the scalar product is lower than valore2 the face is on the second
side.
    elif scalare[i]<=valore2:
        face_list2.append(i)
face_list1=np.array(face_list1)
face_list2=np.array(face_list2)
#using patch_refiner the border pieces are selected
degree=4
face_list1=patch_refiner(mesh,face_list1, scalare,tol, degree, visual)
#the remaining faces are added to the edge
```

```
face_list 3=np.setdiff1d(np.arange(len(mesh.faces)), face_list1)
```

face_list 3=np.setdiff1d(np.arange(len(mesh.faces)), face_list1)
\#using patch_refiner the border pieces are selected
\#using patch_refiner the border pieces are selected
face_list2=patch_refiner(mesh, face_list2,-scalare,tol, degree, visual)
face_list2=patch_refiner(mesh, face_list2,-scalare,tol, degree, visual)
\#the remaining faces are added to the edge
\#the remaining faces are added to the edge
face_list 3=np.setdiff1d(face_list3, face_list2)
face_list 3=np.setdiff1d(face_list3, face_list2)
if len(face_list3)<20:\#if the remaining parts are too small thay can be
if len(face_list3)<20:\#if the remaining parts are too small thay can be
discardeda and no edges are produced
discardeda and no edges are produced
print('No edges')
print('No edges')
return lati, bordi, spigolo
return lati, bordi, spigolo
\#graph of the whole mesh
\#graph of the whole mesh
graph_tot = nx.Graph()
graph_tot = nx.Graph()
graph_tot.add_edges_from(mesh.face_adjacency)
graph_tot.add_edges_from(mesh.face_adjacency)
\#connectivity graph of the edge part
\#connectivity graph of the edge part
mesh3=mesh.copy()
mesh3=mesh.copy()
mesh3.update faces(face list3)
mesh3.update faces(face list3)
graph3 = nx.Graph()
graph3 = nx.Graph()
graph3.add_edges_from(mesh3.face_adjacency)
graph3.add_edges_from(mesh3.face_adjacency)
groups3 = nx.connected_components(graph3)
groups3 = nx.connected_components(graph3)
lista 3=list(groups3)
lista 3=list(groups3)
\#if small pieces are left on the edge part they are attached to the closest
\#if small pieces are left on the edge part they are attached to the closest
border piece. To decide which one is closer the neighbors are found and
border piece. To decide which one is closer the neighbors are found and
compared wih the borders faces.
compared wih the borders faces.
avanzi=[]
avanzi=[]
quanti=0
quanti=0
for element in lista3:
for element in lista3:
element=list(element)
element=list(element)
vicini=[]
vicini=[]
if len(element)<200:
if len(element)<200:
for i in element:
for i in element:
l=list(graph_tot[face_list3[i]])
l=list(graph_tot[face_list3[i]])
l=np.setdiff1d(l, face_list3)
l=np.setdiff1d(l, face_list3)
if len(l)>0:
if len(l)>0:
vicini.extend(l)
vicini.extend(l)
if len(np.intersect1d(face_list1, vicini))}>0\mathrm{ :
if len(np.intersect1d(face_list1, vicini))}>0\mathrm{ :
face_list1=np.append(face_list1,face_list3[element])
face_list1=np.append(face_list1,face_list3[element])
else:
else:
if len(np.intersect1d(face_list2, vicini))}>0\mathrm{ :
if len(np.intersect1d(face_list2, vicini))}>0\mathrm{ :
face_list 2=np.append(face_list2, face_list3[element])
face_list 2=np.append(face_list2, face_list3[element])
else:
else:
quanti+=1
quanti+=1
avanzi.extend(face_list3[element])
avanzi.extend(face_list3[element])
face_list 3=np.array (avanzi)
face_list 3=np.array (avanzi)
\#If the remaining portion of the mesh is big enough another patch is created.
\#If the remaining portion of the mesh is big enough another patch is created.
two attempt are made, from two different starting points. the biggest one is
two attempt are made, from two different starting points. the biggest one is
kept, the remaining faces are marked as 'edge2'
kept, the remaining faces are marked as 'edge2'
if len(face_list3)>20:
if len(face_list3)>20:
degree=10
degree=10
tol=0.005
tol=0.005
edge1=patch_refiner(mesh,face_list3, scalare,tol, degree, visual)
edge1=patch_refiner(mesh,face_list3, scalare,tol, degree, visual)
edge2=patch_refiner(mesh, face_list3,-scalare,tol,degree, visual)
edge2=patch_refiner(mesh, face_list3,-scalare,tol,degree, visual)
if len(edge1)>=len(edge2):
if len(edge1)>=len(edge2):
edge2=np.setdiff1d(face_list3,edge1)
edge2=np.setdiff1d(face_list3,edge1)
else:
else:
edge1=edge2
edge1=edge2
edge2=np.setdiff1d(face_list3,edge1)
edge2=np.setdiff1d(face_list3,edge1)
\#once again the discarded part of the mesh is checked to remove small
\#once again the discarded part of the mesh is checked to remove small
disconnected pieces
disconnected pieces
mesh2=mesh.copy()
mesh2=mesh.copy()
mesh2.update_faces(edge2)
mesh2.update_faces(edge2)
graphe2 = nx.Graph()

```
    graphe2 = nx.Graph()
```

```
        graphe2.add_edges_from(mesh2.face_adjacency)
```

        graphe2.add_edges_from(mesh2.face_adjacency)
        groupse2 = nx.connected_components(graphe2)
        groupse2 = nx.connected_components(graphe2)
        listae2=list(groupse2)
        listae2=list(groupse2)
        avanzi=[]
        avanzi=[]
        quanti=0
        quanti=0
        for element in listae2:
        for element in listae2:
            element=list(element)
            element=list(element)
            vicini=[]
            vicini=[]
            if len(element)<200:
            if len(element)<200:
                for i in element:
                for i in element:
                l=list(graph_tot[edge2[i]])
                l=list(graph_tot[edge2[i]])
                    l=np.setdiff1d(1, edge2)
                    l=np.setdiff1d(1, edge2)
                if len (1)>0:
                if len (1)>0:
                vicini.extend(l)
                vicini.extend(l)
                if len(np.intersect1d(face_list1, vicini)) >0:
                if len(np.intersect1d(face_list1, vicini)) >0:
                face_list1=np.append(face_list1, edge2[element])
                face_list1=np.append(face_list1, edge2[element])
                else:
                else:
                    if len(np.intersect1d(face_list1, vicini))}>0\mathrm{ :
                    if len(np.intersect1d(face_list1, vicini))}>0\mathrm{ :
                    face_list2=np.append(face_list2, edge2[element])
                    face_list2=np.append(face_list2, edge2[element])
                else:
                else:
                    edge1=np.append(edge1, edge2[element ])
                    edge1=np.append(edge1, edge2[element ])
            else:
            else:
                quanti+=1
                quanti+=1
                avanzi.extend(edge2[element])
                avanzi.extend(edge2[element])
    if len(edge1)>0:
    if len(edge1)>0:
        bordi.append(sub_mesh(mesh, edge1))
        bordi.append(sub_mesh(mesh, edge1))
    #if something is still remaining is saved as a second edge piece in
    #if something is still remaining is saved as a second edge piece in
    spigolo
spigolo
if len(avanzi)}>0\mathrm{ :
if len(avanzi)}>0\mathrm{ :
spigolo.append(sub_mesh(mesh,avanzi))
spigolo.append(sub_mesh(mesh,avanzi))
\#the borders are added to lati
\#the borders are added to lati
lati.append(sub_mesh(mesh,face_list1))
lati.append(sub_mesh(mesh,face_list1))
lati.append(sub_mesh(mesh,face__list2))
lati.append(sub_mesh(mesh,face__list2))
if visual:
if visual:
fig = plt.figure()\#shows the points on the edge portion
fig = plt.figure()\#shows the points on the edge portion
ax = fig.add_subplot(111, projection='3d')
ax = fig.add_subplot(111, projection='3d')
ax.scatter (mesh.triangles_center[face_list1,0], mesh.triangles_center[
ax.scatter (mesh.triangles_center[face_list1,0], mesh.triangles_center[
face_list1, 1], mesh.triangles__center[face_list1, 2],c='b', label='border1')
face_list1, 1], mesh.triangles__center[face_list1, 2],c='b', label='border1')
ax.scatter(mesh.triangles_center[face_list2,0], mesh.triangles_center[
ax.scatter(mesh.triangles_center[face_list2,0], mesh.triangles_center[
face_list2,1], mesh.triangles_center[face_list2, 2], c='g', label='border 2')
face_list2,1], mesh.triangles_center[face_list2, 2], c='g', label='border 2')
ax.scatter(mesh.triangles_center[face_list3,0],mesh.triangles_center[
ax.scatter(mesh.triangles_center[face_list3,0],mesh.triangles_center[
face_list3, 1], mesh.triangles_center[face_list3, 2],c='y',label='a\ll edge')
face_list3, 1], mesh.triangles_center[face_list3, 2],c='y',label='a\ll edge')
plt.legend()
plt.legend()
plt.show()
plt.show()
if len(face_list3)>20:
if len(face_list3)>20:
fig = plt.figure()\#shows the edge partitioning in detail
fig = plt.figure()\#shows the edge partitioning in detail
ax = fig.add_subplot(111, projection='3d')
ax = fig.add_subplot(111, projection='3d')
avanzi 3=edge1
avanzi 3=edge1
avanzi4=avanzi
avanzi4=avanzi
ax.scatter (mesh.triangles_center[avanzi3,0], mesh.triangles_center[
ax.scatter (mesh.triangles_center[avanzi3,0], mesh.triangles_center[
avanzi3,1], mesh.triangles_center[avanzi3, 2], c='y', alpha=0.2,label='first edge
avanzi3,1], mesh.triangles_center[avanzi3, 2], c='y', alpha=0.2,label='first edge
')
')
ax.scatter (mesh.triangles_center[avanzi4,0], mesh.triangles_center[
ax.scatter (mesh.triangles_center[avanzi4,0], mesh.triangles_center[
avanzi4,1], mesh.triangles_center[avanzi4, 2], c='r', alpha=0.2, label='second
avanzi4,1], mesh.triangles_center[avanzi4, 2], c='r', alpha=0.2, label='second
edge')
edge')
plt.legend()
plt.legend()
plt.show()
plt.show()
return lati, bordi, spigolo

```
return lati, bordi, spigolo
```


## A.3.3 partition_grid_cut

Functions used apply the windowing to the grid cells.

```
import numpy as np
import scipy as sp
import time
import trimesh as tm
from utilities import estimate_matrix
from utilities import homograpy
from utilities import raw_cut
from utilities import cut
from utilities import sub_mesh
#utility function to build the partitioning grid
def grid_maker(pt, size, edges=True,forced_height=0,forced_width=0):
    # INPUT
    # pt: (4,2) array, the corner points of the grid in clockwise order, starting
    from the lower left corner
    # size: float, maximum height or width of a single cell
    # edges: bool, if True vertically divides the first and last colums an
    ulterior time to obtain a more accurate division of the edges.
    # forced_height: int, number of horizontal lines imposed
    # OUTPUT
    # grid: (a,b,2) array, where a is the number of horizontal lines of the grid,
    b the number of vertical lines. Contains the (x,z) coordinates of the knot
    points in the grid.
    if forced_height==0:
            altezza=int(max(np.linalg.norm(pt[0] - pt[1]),np.linalg.norm(pt[2] - pt[3]))/
    size)
    else:
            altezza=int(forced_height)
    if forced_width==0:
            larghezza=int(max(np.linalg.norm(pt[1] - pt[2]),np.linalg.norm(pt[3]-pt[0]))
    /size)
    else:
            larghezza=forced_width
    x_norm=np.linspace(0,1, larghezza)
    if edges:
            x_norm=np.append([0,x_norm[1]/2],x_norm [1:])
            x_norm=np.append(x_norm[: - 1],[(1+x_norm[-2])/2,1.])
    z_norm=np.linspace(0,1, altezza)
    grid=[]
    homograpy_matrix=estimate_matrix(np.asarray([[0, 0],[0, 1],[1, 1],[1,0]]),pt)
    grid=[]
    for z in z_norm:
            row = []
            for x in x_norm:
                row.append ([x,z])
            grid_row=homograpy(homograpy_matrix,row)
            grid.append(grid_row)
    grid=np.array(grid)
    return grid
#Function to apply the first part of the partitioning to mesh. Using the grid as a
        reference the mesh is divided into cells. Those cells will be further refined
    with following functions.
def windowing(mesh,grid,extra):
    # INPUT
    # mesh: trimesh object, the mesh to be divided
```

\# grid: (altezza, larghezza, 2) array, ( $x, z$ ) knots of the grid disposed so that
$[0,0]->$ lower left corner, $[0,-1]->$ lower right corner, $[-1,0]->$ upper left
corner, $[-1,-1]->$ upper right corner.
\# extra: float, amount each cell should be enlarged in each direction to allow
overlapping
\# OUTPUT
\# sub_grid: (altezza, larghezza) array, contains the extracted sub-meshes
disposed with the same orders as their grid position
\# sub_vertices: (altezza, larghezza, 2) array, the same as grid
\# tempo: float, time cost of the function (seconds)
\#initialize starting time
t $3=$ time.time ()
\#the expansion on the cells borders
extra_border=np. array ([]-extra, -extra],[ - extra, extra], [extra, extra], [extra, -
extra] $]$ )
\#for each cell row the corresponding strip is extracted. Building the strips
first and the cell later is much faster than going for the cells directly
because it does not check all the mesh points every time.
strips $=[]$
facelist=list (range(len(mesh.faces)))
for $h$ in range(len (grid) -1 ):
\#the lower cutting line is decided
plane_origin=np. array ([grid $[h,-1,0]+\operatorname{extra}, 0, \operatorname{grid}[h,-1,1]-\operatorname{extra}])$
second_pt=np.array ([grid $[h, 0,0]-$ extra $, 0, \operatorname{grid}[h, 0,1]-$ extra $])$
direction=np.array (second_pt-plane_origin)
plane_normal=np. array (np.array ([direction[2],0, - direction[0]]))
plane_normal=plane_normal/np. linalg.norm(plane_normal)
\#and $\bar{t} h e$ faces abowe it are selected. Those are also the only faces
further analyzed, the une below the cutting line are dcertainly below the
upper cutting lines too.
facelist=raw_cut(mesh, facelist, plane_normal, plane_origin)
\#the upper cutting line is selected
plane_origin=np.array ([grid $[\mathrm{h}+1,0,0]-\operatorname{extra}, 0, \operatorname{grid}[h+1,0,1]+$ extra] $)$
second_pt=np.array ([grid $[h+1,-1,0]+$ extra, $0, \operatorname{grid}[h+1,-1,1]+$ extra] $)$
direction=np. array (second_pt-plane_origin)
plane_normal=np.array (np.array ([direction[2],0, - direction [0]]) )
plane_normal=plane_normal/np. linalg.norm(plane_normal)
\#and the faces below it are selected
facelist_strip=raw_cut(mesh, facelist, plane_normal, plane_origin)
\#the strip is extracted and added to the list
mesh_strip=sub_mesh (mesh, facelist_strip)
strips.append (mesh_strip)
\#print the time necessary for this middle step
t $4=$ time.time ()
\#print ('Time necessary to cut the strips:',(t4-t3))
sub_grid=[]
sub_vertices $=[]$
for $\bar{h}$ in range(len $(\operatorname{grid})-1)$ :
\#print('strip $\left.{ }^{\prime},(h+1)\right)$
row $=[]$
row_coord $=[]$
strip=strips[h]
larghezza=len (grid [0]) -1
for 1 in range(larghezza):
\#select the corners of the cell to be cut
sub_pts=np. $\operatorname{array}([\operatorname{grid}[h, l], \operatorname{grid}[h+1, l], \operatorname{grid}[h+1, l+1], \operatorname{grid}[h, l+1]])$
if $\bar{l}==1$ :\#the second column should not expand to the left to avoid
including the edges
sub_pts=sub_pts+np. array ([[0, - extra],[0, extra], [extra, extra], [
extra,-extral])
 the right to avoid including the edges sub_pts=sub_pts+np. array ([[-extra, -extra],[-extra, extra], [0, extra ],[0,-extra]])
elif $l==0: \# t h e ~ l e f t ~ e d g e ~ s h o u l d ~ b e ~ l e s s ~ e x p a n d e d ~ o n ~ t h e ~ r i g h t ~ t o ~ k e e p ~$ the borders small
sub_pts=sub_pts+np.array ([[-extra, -extra],[-extra, extra], [extra/2,
extra],[extra/2,-extra]])
elif l==(larghezza-1):\#the right edge should be less expanded on the
left to keep the borders small
sub_pts=sub_pts+np.array ([[-extra/2,-extra],[-extra/2, extra], [
extra, extra], [extra, -extra]])
else:\#all the other cell an be expanded without issues
sub_pts=sub_pts+extra_border
\#the corners and the sub- mesh are added to the lists representing
that row
row_coord.append (sub_pts)
window=cut (strip, sub_pts)
row. append (window)
\#the lists are attached to keep the data structure similar to the grid
row=np.array (row)
row_coord=np. array (row_coord)
sub_grid. append (row)
sub_vertices.append (row_coord)
sub_grid=np. array (sub_grid)
sub_vertices=np. array (sub_vertices)
t4=time.time()
tempo $=(\mathrm{t} 4-\mathrm{t} 3)$
\#print('Time necessary to make the cuts along the xz plane:',tempo)
return sub_grid, sub_vertices, tempo

## A.3.4 utilities

Utilities functions, mainly necessary to import the data and build sub-meshes.

```
import numpy as np
import scipy as sp
import matplotlib
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import networkx as nx
import trimesh as tm
#This function build the matrix used in the aproximation, each row is [1, x, x
    ^2,\ldots., x^ degx,y*1,y*x,\ldots, y^degy*x^degx]
#for each (x,y) coordinate given as input((n,2) list of some kind). degx is the
    maximium degree of x,degy the maximum
#degree of y.
#Returns a (n,(degx +1)*(degy+1)) matrix object(n=number of points given).
def matricexy(xy, degx, degy):
    #INPUT
    # xy: (n,2) array, x and y coordinates of points to be approximated
    # degx: int, degree of the polynomial along the x axis
    # degy: int, degree of the polynomial along the y axis
    #OUTPUT
    # A: (n, number of monomial) matrix, i-th row is [1,x,x^ 2,\ldots, x^degx,y*1,y*x
    ,\ldots., y^degy*x^degx] for (x,y)=xy[i]
    x=np.asarray (xy[:,0])
    y=np.asarray (xy[:, 1])
    A=np.zeros}((\operatorname{len}(x),(\operatorname{deg}x+1)*(\operatorname{deg}y+1))
```

```
    for ey in range(degy+1):
        for ex in range(degx+1):
            A[:, ey *(degx+1)+ex]=np.multiply(x**ex,y**ey)
    return A
#This function evaluate a polynomial of degree degx in x and degy in y with
    parameters p consistent with matricexy
#function over the coordinates given in a (n,2) list or array. returns a list (n
    ,1) with such values.
def poly_value(xy,p,degx, degy,A=[]):
    #INPUT
    # xy: (n,2) array, x and y coordinates of points to be approximated
    # p: array of floats with the polynomial parameters values relative to [1,x,x
    ^ 2,\ldots., x^degx y y*1,y*x,\ldots., y^degy*x^degx]
    # degx: int, degree of the polynomial along the x axis
    # degy: int, degree of the polynomial along the y axis
    # A: (n, number of monomial) matrix, i-th row is [1,x,x^2,\ldots,, x^degx,y*1,y*x
    ,\ldots., y^degy*x^degx] for (x,y)=xy[i]
    #OUTPUT
    #z_value: (n,) array, polynomial with parameters p evaluated values over the
    xy corrdinates
    x=np.asarray (xy[:,0])
    y=np.asarray (xy[:, 1])
    #if A is not given evaluates it
    if A==[]:
        A=np.zeros((len(x),(degx +1)*(degy+1)))
        for ey in range(degy+1):
            for ex in range(degx+1):
                A[:, ey *(degx+1)+ex]=np.multiply(x**ex,y**ey)
    z_value=np.dot(p,A.T)
    return z_value
#Function used to estimate the homographyc transformation matrix necessary to map
    src(source) into dst(destination) points. Will be used to map the mesh points
    into the unit square to ease the spline construction.
def estimate_matrix( src, dst):
    """Estimate the transformation from a set of corresponding points.
    You can determine the over-, well- and under-determined parameters
    with the total least-squares method.
    Number of source and destination coordinates must match.
    The transformation is defined as::
        X=(a0*x+a1*y+a2) / (c0*x + c 1 *y + 1) 
    These equations can be transformed to the following form::
        0 = a 0*x + a 1*y + a2 - c0*x*X - c1*y*X - X
    0 = b0*x + b1*y + b2 - c0*x*Y - c1*y*Y - Y
    which exist for each set of corresponding points, so we have a set of
    N * 2 equations. The coefficients appear linearly so we can write
    A x = 0, where::
        A = [l[\begin{array}{lllllll}{\textrm{x}}&{y}&{1}&{0}&{0}&{0}&{-x*X - y*X -X]}\end{array}]
                        [0 0 0 0 x y 1 -x*Y -y*Y -Y]\ldots...]
        x.T = [a0 a1 a2 b0 b1 b2 c0 c1 c3]
    In case of total least-squares the solution of this homogeneous system
    of equations is the right singular vector of A which corresponds to the
    smallest singular value normed by the coefficient c3.
    In case of the affine transformation the coefficients c0 and c1 are 0.
    Thus the system of equations is::
        A = [[lllllllll
            [0 0 0 x y 1 -Y]...]
    x.T =[a00 a1 a2 b0 b1 b2 c3]}
```

```
    Parameters
    src : (N, 2) array
        Source coordinates.
    dst : (N, 2) array
    Destination coordinates.
Returns
    - V[-1, :-1]/V[-1,-1] : (8,1) array
        Parameters of the transformantion
    """
    xs}=\operatorname{src}[:, 0
    ys}=\operatorname{src}[:, 1
    xd = dst[:, 0]
    yd = dst[:, 1]
    rows = src.shape[0]
    # params: a0, a1, a2, b0, b1, b2, c0, c1
    A = np.zeros((rows * 2, 9))
    A[:rows, 0] = xs
    A[:rows, 1] = ys
    A[:rows, 2] = 1
    A[:rows, 6] = - xd * xs
    A[:rows, 7] = - xd * ys
    A[rows:, 3] = xs
    A[rows:, 4] = ys
    A[rows:, 5] = 1
    A[rows:, 6] = - yd * xs
    A[rows:, 7] = - yd * ys
    A[:rows, 8] = xd
    A[rows:, 8] = yd
    ## Select relevant columns, depending on params
    #A = A[:, list(self._coeffs) + [8]]
    _, _, V = np.linalg.svd(A)
    H=np.zeros((3, 3))
    # solution is right singular vector that corresponds to smallest
    # singular value
    H.flat [: -1] = - V[-1, : - 1] / V[-1, -1]
    H[2, 2] = 1
    return - V[-1,: - 1]/V[-1, -1]
#Utility function to apply the homograpy to a set of points
def homograpy(v, points):
    #INPUT
    # v: (8,) array, transformation parameters
    # points: (n,2) array, starting points
    #OUTPUT
    # np.vstack([X,Y]).T: (n,2) array, transformed points.
    points=np.asarray(points)
    [a0, a1, a2, b0, b1, b2, c0, c1] =v
    x=points[:,0]
    y=points[:,1]
    X = np.asarray ((a0*x + a 1*y + a2) / (c0*x + c1*y + 1))
    Y=np.asarray ((b0*x + b1*y + b2) / (c0*x + c1*y + 1) )
    return np.vstack([X,Y]).T
#This function find the faces of the mesh with at least one vertex over the plane
    selcted, with respect to the plane normal. To do this all the points are
```

```
    translated to a new system of coordinates centered in plane_origin, with a
    simple substraction. Then the pints are projected (as vectors) against the
    plane normal. If at least one of the vertices of a face gives a positive
    result the face is added to faces_list. The parameter faceslist gives the list
    of faces of the mesh that must be considered, this is useful in particular
    when we want to do more than one cut. If we want to cut a portion of the mesh
    we apply this function for each edge of the partition, with faceslist equal to
    the result of the previous cut. This way we scrape off a portion of the mesh
    each time, the process get obviously faster each time.
def raw_cut(mesh, faceslist, direction, plane_origin):
    #INPUT
    # mesh: trimesh object
    # facelist: array or list object, contains the faces indexes to be checked
    # direction: (3,) array, direction normal to the cutting plane
    # plane_origin: (3,) array, point on the plane.
    #OUTPUT
    # faces_list: list object containing the indexes of the mesh faces above the
    plane(intersected facelist)
    #confirms that the type of the data is array, mesh must be a trimesh file and
    faceslist a list or array
    plane_normal=np.asanyarray(direction)
    plane_origin=np.asanyarray(plane_origin)
    #pick the vertices of the desired faces
    face_vertices=mesh.faces[faceslist]
    #compute the projection against the plane normal=direction
    dots = np.dot(plane_normal, (mesh.vertices - plane_origin).T)[face_vertices]
    faces_list=[]
    for ii in range(len(faceslist)):
        #if the projection on the normal vector is positive the point is on the
    right side of the plane
        #replacing any with all gives triangles that are completely over the plane
        #It could be necessary to substitute 0 with some constant to avoid
    numerical error
        if any(dots[ii,:] >= 0) :
            faces list.append(faceslist[ii])
    return faces_- list
#This function applies raw cut along the edges of a quardilateral defined by its
    vertices, disposed clockwise from the lower-left one. If only a portion of the
        faces are of interest that information can be given by facelist. Returns the
    portion cut as a trimesh object.
def cut(mesh,pt, facelist=[]):
    #mesh = trimesh object, the mesh we are going to window
    #pt = (4,2) array, (x,z) coord of the corners of the quadrangle, clockwise
    from the one in the lower left
    # faces_list: list object containing the indexes of the mesh faces above the
    plane(intersected facelist)
    #OUTPUT
    #facelist= array, optional, list of starting faces to consider. Default= all.
    #control to check if all the faces should be used
    if len(facelist)==0:
        facelist=list(range(len(mesh.faces)))
        #once for each side raw cut is applied
    for i in range(4):
        #the plane is selected from the corner points of the quadrilateral cell
        plane_origin=np.asanyarray ([pt[i,0],0,pt[i, 1]])
        #If we are working with the forth point we need to go use the first to
    make it work
```

\#This utility function is used to extract a sub-mesh. It builds a dictionary that
associate the indices of the vertices in the given mesh with the new indices
they will have in a new one with only the triangles in face_list. It also
gives a list of the new vertices ordered with the new indexing. As input it
require a starting mesh as a trimesh object, and faces list as a list or array
The order of the vertices is completely different from the original one, I
have no idea why. The
def from_vertices_to_dict(mesh, faces_list):
\#INPUT
\# mesh: trimesh object
\# facelist: array or list object, contains the faces indexes of the sub-mesh
\#OUTPUT
\#dictionary: dict object, associate each face(from 0 to len(face_list)) to the
correspective vertices index in new vertices
\#new_vertices: (n,3) list of the vertices in the submesh.
\#load the faces as an array
faces=mesh.faces.view (np.ndarray)
count=3
\#the first three vertices(the vertices of the first face) are inserted
manually in both the list and the dictionary
new_vertices=[mesh.vertices[faces[faces_list[0]][0]],
mesh.vertices[faces[faces_list[0]][1]],
mesh.vertices[faces[faces_list [0]][2]]]
dictionary ={faces[faces_list[0]][0]:0, faces[faces_list[0]][1]:1, faces[
faces list[0]][2]:2}
for i in range(1, len(faces_list)):
for j in range(3):
new_vertex=faces[faces_list[i]][j]
if new_vertex in dictionary:
\#if the vertex is already in the dictionary we pass to the next
one
continue
else:
\#otherwise we add a new key(and value) and the new vertex to the
list
dictionary[new_vertex]=count
count+=1
new_vertices.append(list (mesh.vertices[new_vertex]))
return dictionary, new _vertices
\#This function build a new mesh of a subset of the mesh given as a input, with the
faces present in
9 \#faces_sequence(list). The idea is to select only the vertices belonging at(at
least) one of the faces selected,
220 \#re-indexing them and to build a dictionary to associate the old indexes to the
new one. This is helpful to change

```
```

\#the indexes of the faces, and should be useful to reverse the process if
necessary
def sub_mesh(mesh, faces_sequence):
\#INPUT
\# mesh : Trimesh object
\# faces sequence : sequence of face indices from mesh
\#OUTPUT
\# result : Trimesh object, the portion of the mesh
\# check input type
faces_sequence = list(faces_sequence)
\# work on a copy to avoid nuking the cache on the original mesh
original_faces = mesh.faces.view(np.ndarray)
\#build the dictionary and the updated vertices list
dictionary, new_vertices=from_vertices_to_dict(mesh, faces_sequence)
\#Initialize the updated faces list and insert the first one
faces = [[ dictionary[mesh.faces[faces_sequence[0]][0]],
dictionary[mesh.faces[faces_sequence[0]][1]],
dictionary[mesh.faces[faces_sequence[0]][2]]]]
\#Define the updated vertices list and normal vetors
vertices = np.asanyarray(new_vertices)
normals = np.asanyarray (mesh.face_normals[faces_sequence])
for faces_index in faces_sequence[1:]:
\#for each face on the list
faces_current = original_faces[faces_index]
\#we update the indexes of it's vertices
face_updated=[dictionary[faces_current[0]], dictionary[faces_current [1]],
dictionary[faces_current[2]]]
\#And we append the result to the update faces list
faces.append(face_updated)
\#this command build the ne sub mesh
result=tm.base.Trimesh(vertices, faces, normals)
return result
\#This function move the mesh points given so that it's new center of mass became
the origin of the axes. It also apply a rotation to all the points to allign
the surface with the xy plane. The rotation matrix is built up from the
principal direction obtained from PCA of the covariance matrix. As input
require a mesh object and return the rotation and center(to reverse the
process if needed) and the new coordinates new_coord0.
def workbench(mesh,l=[0,1]):
\#INPUT
\# mesh: trimesh object
\# l: optional (2,) list of integers between [0,1,2], they indicate which PC
direction should be used as first and second axis.
\#OUTPUT
\# rotation: (3,3) matrix, rotation matrix
\# center: (3,) array, previous center of the mesh mertices
\# new_coord: (n, 3) array, roto-translated vertices coordinates
\#initialize the points we'll work with
vertici=np.matrix(mesh.vertices)
\#we find the mean point, or center of mass
center=np.array(np.mean(vertici , axis=0))
\#and translate the points
translated=(vertici-center )
\#we build the covariance matrix

```
```

covariance_matrix=np.dot(translated.T, translated)
\#we find the eighenvectors
U, s, V=np.linalg.svd(covariance_matrix)
v1=V[1[0]]
v2=V[1[1]]
\#v3=V[2] \#just to be sure we find the last vector with a cross product to
avoid signs problems
plane_normal=np.cross(v1,v2)
\#and build the rotation matrix
rotation=np.asanyarray ([v1,v2, plane_normal])
rotation=np.matrix(rotation)
\#we define the new coordinates
new_coord=np.asarray(np.dot(rotation, translated.T).T)
\#print(s)
return rotation , center, new_coord

```

\section*{Bibliography}
[1] MTU Aero Engines: https://www.mtu.de/
[2] Damir Vučina, Milan Ćurković, Tonči Novković ; Classification of 3D shape deviation using feature recognition operating on parametrization control points, Computers in Industry 65 (2014) 1018-1031.
[3] https://en.wikipedia.org/wiki/STL_(file_format)
[4] A. Bjorck. Numerical Methods for Least Squares Problems. SIAM, 1996.
[5] I.T.Jolliffe,2002, Principal Component Analysis, Second Edition, Springer
[6] M. S. Floater, How to Approximate Scattered Data by Least Squares, SINTEF Report No. STF42 A98013, Oslo, (1998).
[7] https://en.wikipedia.org/wiki/Wind_turbine_design
[8] http://www.offshorewindindustry.com/news/mhi-vestas-launched-worlds-powerful-wind
[9] M. von Golitschek and L. L. Schumaker, Data fitting by penalized least squares, in Algorithms for Approximation II,J.C. Mason and M.G.Cox (eds.),Chapman \& Hall(London),1990,210227.
[10] James W C, Gorton S H and Alby F, 2005, Effects of Sand and Dust on Small Gas Turbine Engines (NATO Research and Technology Organization)
[11] R Hussin et al, 2016, IOP Conf. Ser.: Mater. Sci. Eng. 152012038
[12] R. Blomley, M. Weinmann a, J. Leitloff, B. Jutzi Shape distribution features for point cloud analysis- A geometric histogram approach on multiple scales. ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume II-3, 2014
[13] S. Fleishman, D. Cohen-Or, and C. T. Silva, Robust Moving Least-squares Fitting with Sharp Features, in ACM transactions on graphics (TOG), 2005, pp.544-552.
[14] C. Mineo, S.G. Pierce, R. Summan, Novel algorithms for 3D surface point cloud boundary detection and edge reconstruction, Journal of Computational Design and Engineering (2018)
[15] The MIT License (MIT), Copyright (c) 2015 Michael Dawson-Haggerty, Trimesh Python library, https://github.com/mikedh/trimesh, license detail on https://github.com/ mikedh/trimesh/blob/master/LICENSE.md
[16] Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, Exploring network structure, dynamics, and function using NetworkX, in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11-15, 2008
[17] Splipy python libray, https://github.com/sintefmath/Splipy, license detail on https: //github.com/sintefmath/Splipy/blob/master/COPYING
[18] B.V.R Ravi Kumar, A review on blisk tecnology,International Journal of Innovative Research in Science, Engineering and Technology, Vol. 2, Issue 5, May 2013.
[19] Piegl, Les A., and Wayne Tiller; The NURBS book. Berlin: Springer, 1997.
[20] Yang Fengna, Pan Chengxiong, Zhang Dongfang, Tang Jian and Yan Jing,Stress distribution and deformation analysis of gas turbine blades and disk with FEM method,Proceedings of the ASME 2017 Power Conference Joint With ICOPE-17 POWER2017-ICOPE-17 June 26-30, 2017, Charlotte, North Carolina, USA
[21] Seyed Masoud Marandi, Khosrow Rahmani, Mehdi Tajdari, Foreign object damage on the leading edge of compressor blades, transaction of Famena XXXVII-2 (2013)
[22] Tim J. Carter, Common failures in gas turbine blades, Engineering Failure Analysis 12, 237-247, 2005
[23] Xi Chen, Foreign object damage and fatigue cracking. PhD Thesis, Division of engineering and applied science department of Harvard University, (2001).
[24] https://www.gom.com/industries/power-generationpropulsion.html
[25] https://www.olympus-ims.com/en/applications/rvi-passenger-jet-engine
[26] https://www.gom.com/

\section*{Acknowledgements}

First of all I would like to thank MTU Aero Engines [1] for providing me with the opportunity to work on this topic. It has been an extremely instructive experience, that enriched me in many ways.

In particular, I want to thank my supervisors Dr. Giulia Antinori and Julian von Lautz. They consistently allowed this paper to be my own work, but steered me in the right direction whenever I needed it.

I also want to thank Prof. Fabio Marcuzzi and the mathematics department of the university of Padua.

Finally, I must express my very profound gratitude to my family and to my friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis.

This accomplishment would not have been possible without them. Thank you.```


[^0]:    ${ }^{1}$ There are also other types of damages possible, like cracks, but will not be discussed in this work because their size was too small to be detected by the scanner. Given more accurate measurements the same approach proposed should still be usable to detect them, with different search parameters, as shown in [2].

[^1]:    ${ }^{1}$ A third kind of damage they consider are cracks or fissures, in this case the second components is expected to be sensibly smaller than the first one.

[^2]:    ${ }^{1}$ Further test has been performed with higher degrees but did not show any significant difference in the results.

[^3]:    ${ }^{1}$ In fact many extra findings where simply double or triple marking on parts of the same damage because of its position across multiple patches.

[^4]:    ${ }^{1}$ Additional parts not included in this list could be added if necessary as long as the patches extracted have a structure simple enough.

[^5]:    ${ }^{2}$ while considering only the $(y, z)$ coordinates of the PCA-oriented points, namely projecting along the edge length.

