Motion Planning and Image Capturing for Robotic Inspection of a Curved Surface Subject to Imaging Constraints

Abhishek Arun Kulkarni

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science

University of Washington

2020

Committee:

Xu Chen

Santosh Devasia

Sawyer B. Fuller

Program Authorized to Offer Degree:

Mechanical Engineering

University of Washington

Abstract

Motion Planning and Image Capturing for Robotic Inspection of a Curved Surface Subject to Imaging Constraints

Abhishek Arun Kulkarni

Chair of the Supervisory Committee:

Professor Xu Chen

Department of Mechanical Engineering

Quality Control is an important step in manufacturing of machine parts especially for complex, customized parts for safety critical systems like airplane engines. Visual Inspection by humans is one of the modalities used for this purpose, but it limited in many ways. Inspectors need months of training, need to maintain tremendous focus for a long duration, and are required to keep up with the growing pace of manufacturing. It is thus imperative to automate this process. This thesis proposes a methodology for automated inspection by using a commercial robot arm (Universal Robots UR5e), describes elements of the solution, and evaluates its efficacy – pertaining to complex metallic parts with characteristics similar to components of a jet engine. Furthermore, this thesis also explores a potential improvement to the automated inspection process which implements a mesh segmentation algorithm and attempts to generalize the custom solution.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am extremely grateful to my adviser Prof. Xu Chen who provided me the opportunity to contribute to the research work done in University of Washington's MACS Lab. The time spent in the lab has not only made me a better engineer, but also an effective communicator. I thank Prof. Santosh Devasia and Prof. Sawyer Fuller for serving on my committee and for expanding my knowledge on Robotics and Control Theory.

I would also like to thank fellow lab members Dan Wang and Hui Xiao for guiding me at different stages of my project. Finally, I am forever grateful to my friends and family for their constant support.

DEDICATION

to my father and mother, Arun and Anuja Kulkarni

1. INTRODUCTION

1.1 Background and Related Work

In the context of engineering, “Inspection” can be defined as the comparative evaluation of an object or activity, which involves measurements, tests, and gages. It can be classified into various methods – particularly relevant to the scope of this text is visual inspection. It can be used to detect a wide range of surface defects like dents, scratches, corrosion, discoloration, and others [1]. Various factors, for example, the critical nature of the inspected object or the process, the volume of manufactured parts, and the industry application affect the standards with which the inspection process is carried out. Inspections can be conducted on a fraction of all manufactured parts (called batch inspection) or 100% of the parts. Literature suggests that batch inspection conducted by humans needs to be significantly more accurate than total inspection, and the costs associated with total inspection are significantly higher [2]. Considering the Aerospace Industry for instance, it relies more on total inspection due to its mission critical nature and human safety requirements. Overlooking minor defects like scratches and dents even in one of the critical parts (like a stator vane) could lead to large imbalances in airflow, premature fatigue failure, and even jeopardize human life in some cases. The aerospace manufacturing industry is growing, but the rigorous and necessary inspection pipeline limits the throughput.

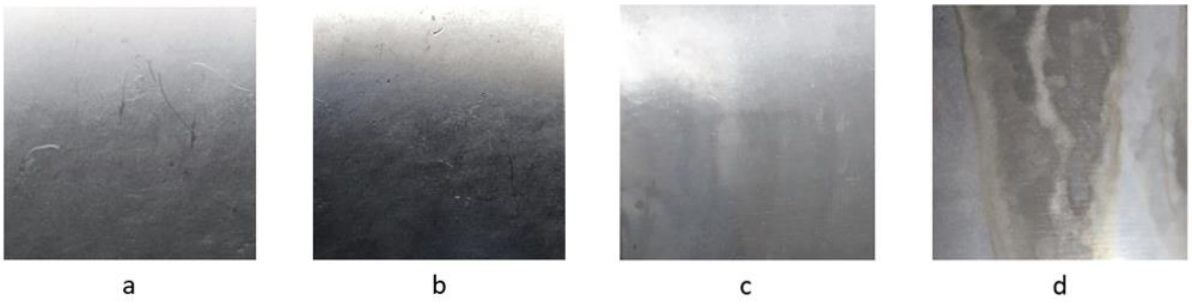

Figure 1.1: Different defects on a metallic surface – (a) Scratches, (b) Dents, (c) Discoloration, (d) Corrosion

There is an extensive amount of research in the past that documents the methodologies and setups for inspection pipelines applied to various problems [3]. The problem of defect detection and classification using computer vision algorithms is not a new one – literature dated in the 1990s can be found, which employs image processing algorithms to detect cracks and corrosion on aircraft skin, by using wavelet transform [4].

Various other strategies that apply image processing algorithms (like morphological operations, histogram processing) for metallic surfaces are documented as well [5]. One relevant method is to analyze grey level co-occurrence matrices to determine the type and location of anomalies in the texture of a metallic part [6]. Research has also been conducted on a process to image extremely reflective metal parts, and classify its defects by implementing SVM. This work also offers insight on the lighting conditions that are conducive to image acquisition [7]. There are many custom robotic platforms that serve as a partial or total inspection solution for specific applications like underwater vessel hull maintenance [3]. The arguments made in favor of such appliances can be extended to aerospace inspection industry as well.

1.2 Motivation

Consider the stator vane of a jet engine. It is a highly customized, complex, metallic part with a very critical role in the aircraft. Faults in a stator vane can potentially lead to disastrous consequences, that is why they undergo total inspection as opposed to batch inspection. It is a high production volume part, and an average manufacturing plant can produce 500 such vanes per day. It can be deduced that the inspection process is a bottleneck for the output of a plant, and with increasing demand, the inspection load is ever increasing. Currently, humans are employed for this task and require years of training to master the trade – each vane must conform to stringent standards and thus, visual inspection needs to be error free.

Surface inspection is arguably a monotonous job. It also requires advanced instruments and sharp focus for extended hours, which is tiring and affect objectivity in decision making, leading to inaccurate classification of parts. With increase in demand for personnel and lack of supply due to an increasing trend of automation, it has become imperative to move to a more sustainable and long-term solution. It has been established that machine learning principles can be applied to the

classification problem, and we also have sophisticated hardware to do so. If a solution that combines principles of lighting, state of the art machine learning, image processing, and industrial robotics is implemented, it will cut manufacturing costs and increase production volume as well.

1.3 Contribution

The main idea of the proposed solution is to use a robotic arm to manipulate parts, and use a high definition DSLR camera to capture images of the part. These images after processing will be fed to a classifier network which advises on the defect type and location. The entire system will be controlled in ROS (Robot Operating System) environment.

The contribution of this thesis is two-fold:

- Contribute to building certain elements of an advisory robotic platform for robotic inspection of curved metallic parts – the overall procedure, image processing, and lighting setup. As a test subject for validation of the system, a stator vane was selected owing to its high production volume.
- Extend the capabilities of the inspection cell to a general 3D part. Given the 3D mesh of a generic test subject, this thesis attempts to implement an algorithm to determine waypoints for a camera mounted robotic system for total imaging of the part surface.

Since the goal is to design specific elements of the solution, gaps have been filled with explanations wherever necessary. To limit the scope of this project, certain assumptions are made. For example, in the generalized algorithm, the geometry of the part is assumed to be convex. Certain elements of the solution are primitive because they are implemented with the aim of providing a Proof of Concept. Sophisticated versions of these elements will be included in the future. Finally, the stator vane used as test piece is an export-controlled part and its illustrations will be partial or be represented by a similar component wherever necessary.

1.4 Organization of Dissertation

This dissertation comprises of two main parts, which covers two phases of the project. In the first phase, a working solution is described for the inspection problem and in the second phase, a

possible improvement to that solution is explored. Chapter 2 - “Manual Waypoint Recording” describes the developed process which can be used to solve the given problem satisfactorily, and which fulfilled the project deliverables. The different elements of the process are described in detail. In Chapter 3, “Automated Waypoint Generation”, the key highlights of the proposed improvement are explained in detail. In Chapter 4 - “Experiments and Simulation”, the entire inspection process is simulated in a physics engine, which provides a realistic view of the pipeline. Images captured by the Manual Waypoint Recording Process are presented, and both strategies (Chapter 2 and 3) are evaluated heuristically as well. Finally, in Chapter 5, the limitations are addressed, and possible future work is explored.

2. MANUAL WAYPOINT RECORDING

2.1 Hardware Setup

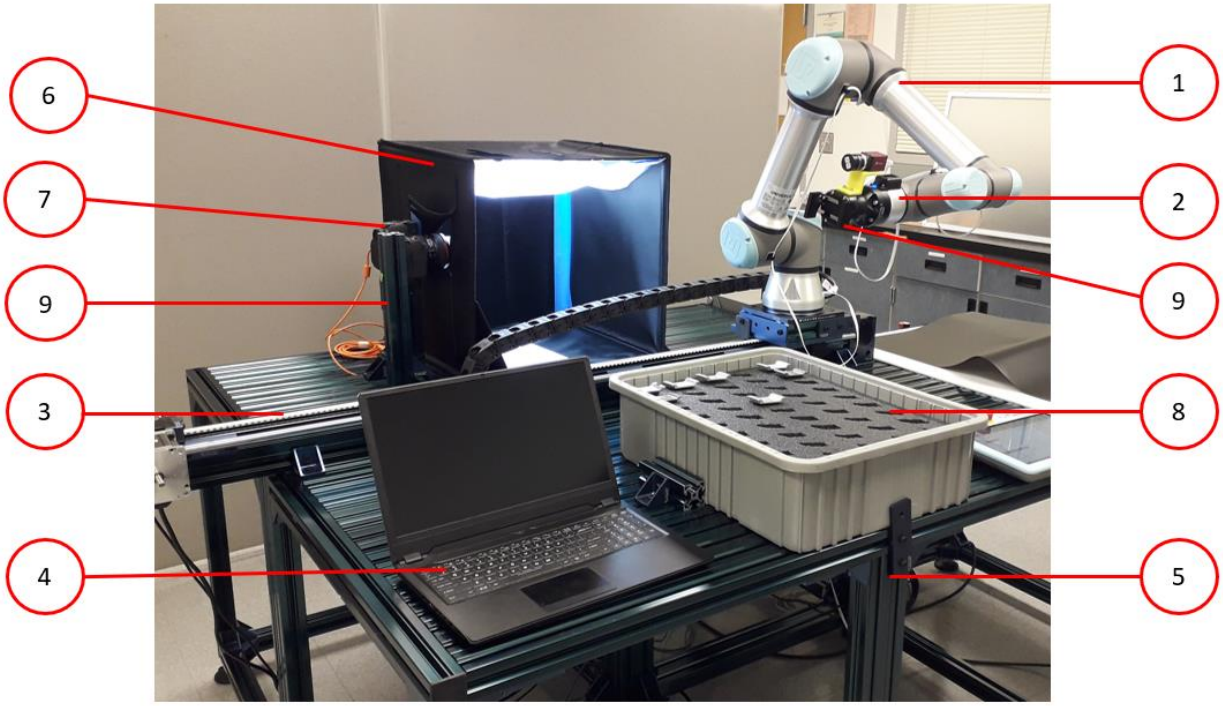

Figure 2.1: Setup for Manual Waypoint Recording – (1) UR5e Robot, (2) Hand-e Gripper, (3) Linear Stage, (4) Computer, (5) Base, (6) Shooting Tent, (7) DSLR Camera, (8) Bin with Foam Insert, (9) Gripper Fingers and Camera Mount

The Inspection Cell consists of various off-the-shelf and custom designed components that contribute to the process. The major parts are illustrated in figure 2.1 and their descriptions are mentioned in the following:

1. Universal Robots UR5e – A 6-DOF robot arm, used for manipulating the stator vanes. A "Teach Pendant" provided by Universal Robots can be used to control the arm and program trajectories for the robot.
2. Robotiq Hand-E Adaptive Gripper – Used to hold the parts. The stock fingers are not used, and custom design is fabricated

3. Linear Timing Belt Actuator (called Linear Stage) – A 1-DOF linear motion generator on which the UR5e is mounted.
4. Computer – A desktop computer with Linux Operating System.
5. Base – A custom designed workbench which will be used to mount other components like camera, bins that contain the parts to be inspected, etc.
6. Shooting Tent – Ideally, it is a structure which provides an isolated environment for imaging the stator vanes. For development phase of the project, an off-the-shelf photographic shooting tent is used which has provisions to insert a camera and overhead lighting with a diffuser fabric. The inner walls are covered with a matte surface for light absorption.
7. Canon EOS 6D Mark ii DSLR Camera with Canon EF 24-70 USM Lens – The camera used for taking high definition images of the parts. It is controlled by a PC using USB interface.
8. Storage Bins with Foam Insert – This is used to hold multiple parts together, and the foam insert is used to secure all the vanes in place inside the bin.
9. Gripper Fingers and Camera Mount – Custom designed to be integrated with other components. Gripper is designed to mechanically restrict the movement of the part in all 6 degrees of freedom which ensures repeatability in holding different parts.

Please note that stator vanes are export controlled and all imagery in the dissertation illustrating these parts will be partial. Some preliminary settings (for example, camera parameters) are left out here and will be addressed in the following sections.

2.2 Process

Stator vanes are compact, extremely complex, metal parts that have high reflectivity. It is very difficult to capture images of metallic surfaces without the occurrence of specular reflections. Moreover, high quality images are needed to classify submillimeter size defects. Sharp images of defects are possible only if the camera is focused on specific regions of the surface of the part, and an added requirement is that the surface being imaged should lie within the Depth of Field (extent of focus in front of the camera) limits of the camera. All these constraints would be respected if the area being photographed is small. That is why, in this process, the surface of the stator vane is

divided into smaller sub-regions manually by a human. It is divided in such a way that each sub-region is a quadrangle, and it spans a relatively flat region. Care is taken so that all subregions must have an area which ensures that their span in the 3rd dimension (due to the curvature) should not exceed the depth of field of the camera. For the given stator vane, it results in a total of 23 sub-regions on the entire surface as seen in figure 2.3. The division is done using a masking tape as a pre-processing step; this part will henceforth be referred to as the Calibration Part.

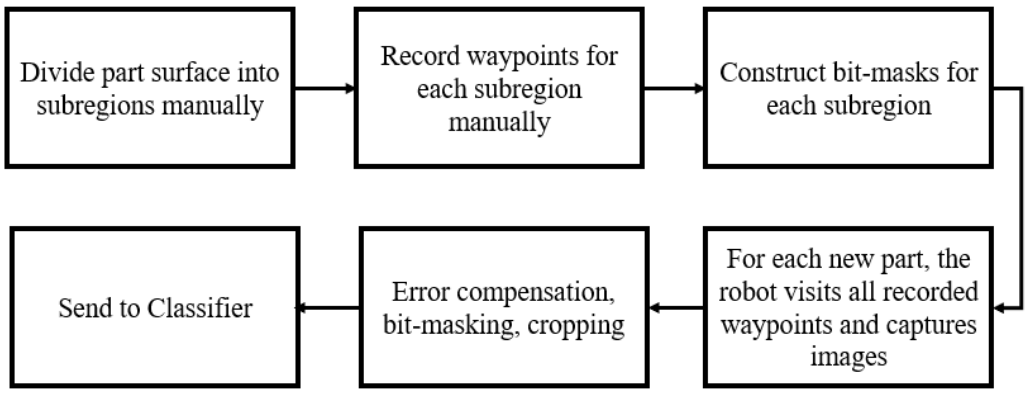

Figure 2.2: Process Flow of Manual Waypoint Recording

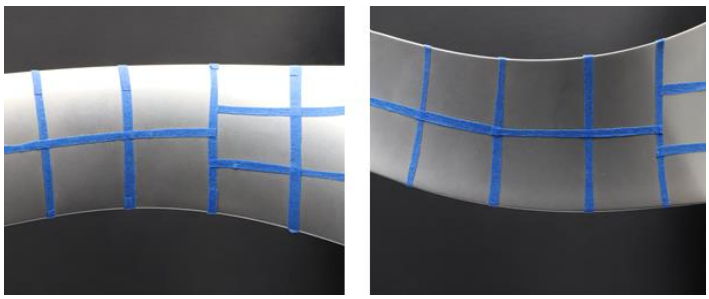

Figure 2.3: Manual Subdivision of the curved metallic part into 23 subregions

After the part is secured into the custom-made gripper by its base, the UR5e follows a predetermined trajectory to photograph the entire part. This trajectory consists of certain waypoints. To image each subregion effectively and to ensure uniform sharpness throughout the

photographed area, the direction of the mean local normal of the sub-region should coincide with the optical axis of the camera as shown in figure 2.4. Additionally, the perpendicular distance of the sub-region should be approximately equal to the focal length so that the curved sub-region lies inside the depth of field. We can calculate that distance using a standard camera model by using the values of aperture and focal length in the camera parameter tuning phase. The human manually sets each waypoint by using the Teach Pendant of the UR5e, by observing the resulting images in the Remote Operation Software and without tweaking camera parameters.

After the UR5e completes its trajectory, the images are processed to suit the needs of the classifier. Due to the large field of view (horizontal and vertical extent of camera viewpoint) of the camera, small depth of field, and the highly curved surface of the stator vane, most regions in the images are out of focus. To extract the correct sub-region, the images need to be cropped to exclude the background, and this process also needs to be automated. For each image of the calibration part, an equivalent bit-mask image is made using an image editing software. The role of the bit-mask image is to extract the desired area of the image and fill the background with black. This technique is explained in detail in the following section. Ultimately, the resulting bit-masked image contains the respective subregion in sharp focus and a black background. This is followed by using a bounding box to crop out the black background, and finally, resizing the image to 500 × 500 pixels. Resizing does result in loss of information, but the remaining pixels that represent defects are enough for the classifier. The mean size of the defect that needs to be detected translates to 11×11 pixels in image space. Experiments from colleagues show that resizing does not affect the classifier's capabilities.

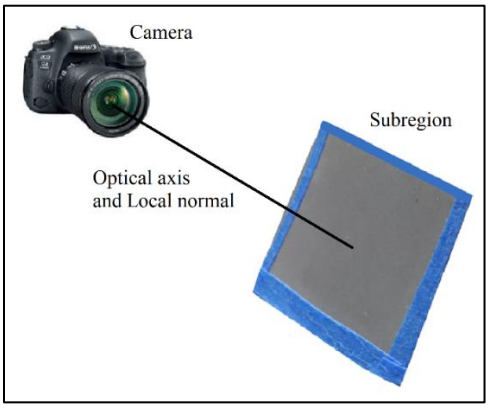

Figure 2.4: Illustration of the relative configuration of the camera and any Subregion

2.3 Bit-Masking

This step is part of the image post-processing phase. After an image is obtained for a sub-region, the desired region of interest in the image must be extracted automatically. For this purpose, a customized bit-mask is created by the human for each subregion, based on the manually segmented surface of the part. The bit-mask is a 3-channel image, with all background pixels black (logical 0) and all foreground pixels white (logical 1). When this image is used on any sub-region with an AND operation, it retains the foreground and marks the background in black. The construction of a bit-mask for each subregion is considered as a pre-processing step, and follows the manual waypoint recording step. The operation is illustrated in figure 2.5.

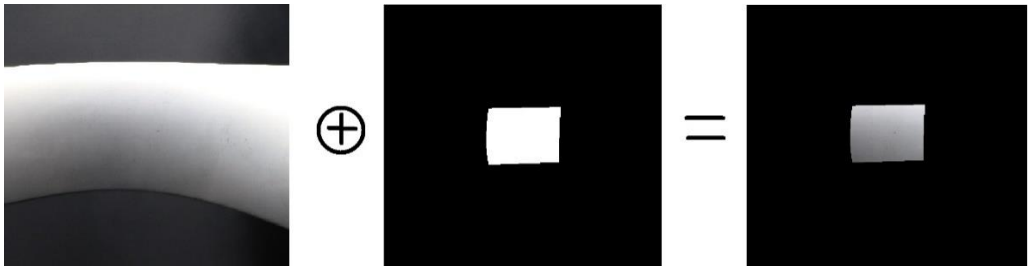

Figure 2.5: Illustration of the bit-masking process – (left) Input, (center) Mask, (right) Output

There are, however, certain limitations of this technique. Due to the manual nature of waypoint recording, and the fact that the waypoints never change throughout the experiment for parts following the calibration part, the robot follows the prescribed trajectory in an open-loop manner. That is, there is no correction for minor errors that might arise in certain instances of the experiment. Errors can occur in all forms and sources can be any of the following – backlash in robot joint angles, backlash in linear stage of the robot, inconsistency in the relative position of gripper and stator vane, to name a few. As mentioned in section 2.1, even if the gripper is designed to be repeatable in its operation, a software-based solution is necessary to compensate for errors that are not in our control.

Hence, before bit-masking, the image needs to be adjusted to match the position and orientation of the calibration part. For this purpose, tools from OpenCV, a Python library are used. Using feature matching, the relative 2D transform between the calibration image and the current image

is determined, and this is applied to the image. It takes care of the errors in 3 degrees of freedom – two translations in the image plane and one rotation about the optical axis. Corrections in the remaining three degrees of freedom have very little effect on the image and can be ignored. The results are depicted in figure 2.6, and the code is attached in the Appendix.

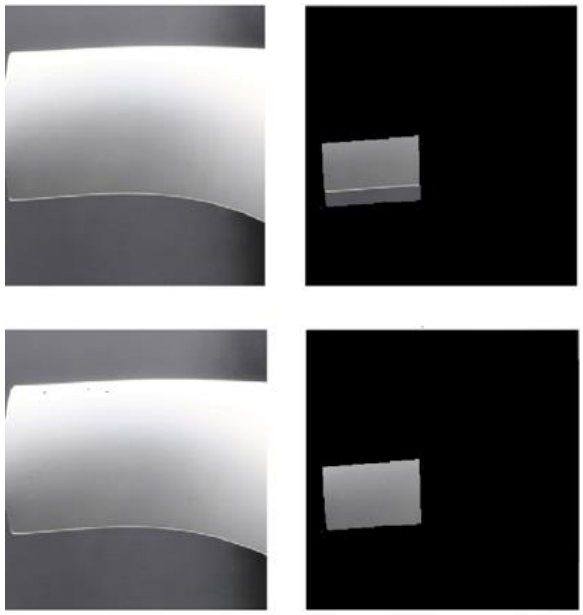

Figure 2.6: Illustration of the bit-making correction step – (top left) Input Image with error in position and orientation, (top right) Output after bit-masking, (bottom left) Corrected Image, (bottom right) Improved Output from bit-masking

2.4 Lighting

Specular reflection is the situation where incident light beam remains coherent before and after reflection. It becomes imperative to manipulate lighting to avoid it during imaging, because the intense light on the optical sensor array in the camera exceeds the upper threshold of its sensitivity capacity and it is rendered as a white spot. This is loss of information in images, and no amount of camera parameter tuning will eliminate it as seen in figure 2.7. One way this glare can be mitigated is by using polarizing filters, which block all light rays of a given orientation [8]. Empirical evidence shows that the image quality (color rendering, intensity) suffers if polarizers are used (see figure 2.8). Moreover, the effect of polarizers is not uniform – it is highly sensitive to the part orientation and the lighting. For these reasons, an alternate solution to lighting is needed.

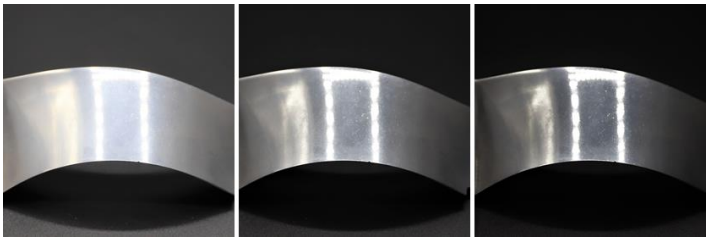

Figure 2.7: Varying camera parameters having no effect on the glare

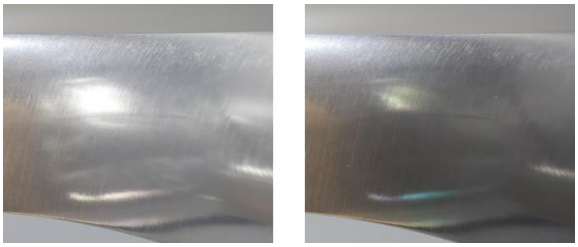

Figure 2.8: Using polarizing filters – (left) Unpolarized, (right) Polarized

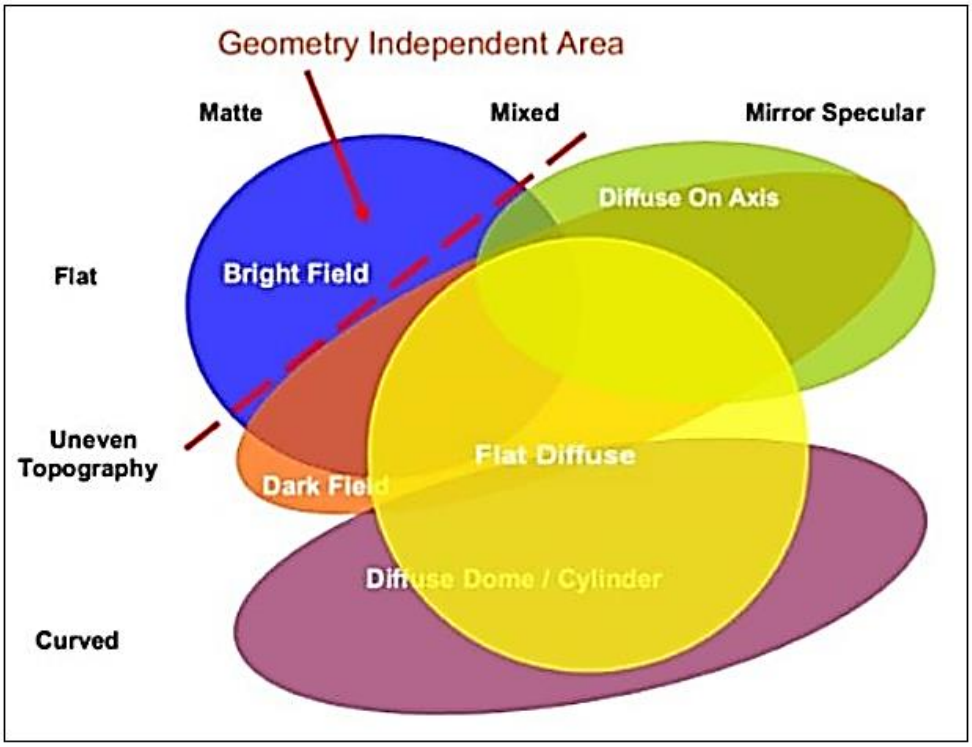

Figure 2.9: A chart depicting lighting configuration for metallic objects based on geometry and surface finish [9]

Imaging of metallic parts can be done in several ways. Many examples can be found with its own set of advantages. Back lighting is used to detect gaps in the object or to facilitate measurement. Dark field lighting is useful to extract textures on surfaces. Diffused lighting is best suited for capturing metallic surfaces, because diffused light is incoherent and hence cannot result in specular reflections [9]. The light is white so that true colors are captured, which in turn results in a more uniform dataset for the training of the classifier. Lighting must respect the "family of angles" criteria. It depends on the relative position of the camera and the imaged surface, and is defined as the range of angles within which if the light source is placed, it causes direct reflection. That is why a Shooting Tent that has an array of white LED lights on the roof, with a diffuser cloth wrapped on it, and its inner walls covered with light absorbing material is a near-perfect setup for lighting. It is compact, simple, and built for photographic applications. The LED array of the Shooting Tent is set to its maximum output, which is 21600 lumens. It is necessary to maximize incident light on the surface. The possibility of overexposure can be eliminated by adjusting the ISO setting on the camera.

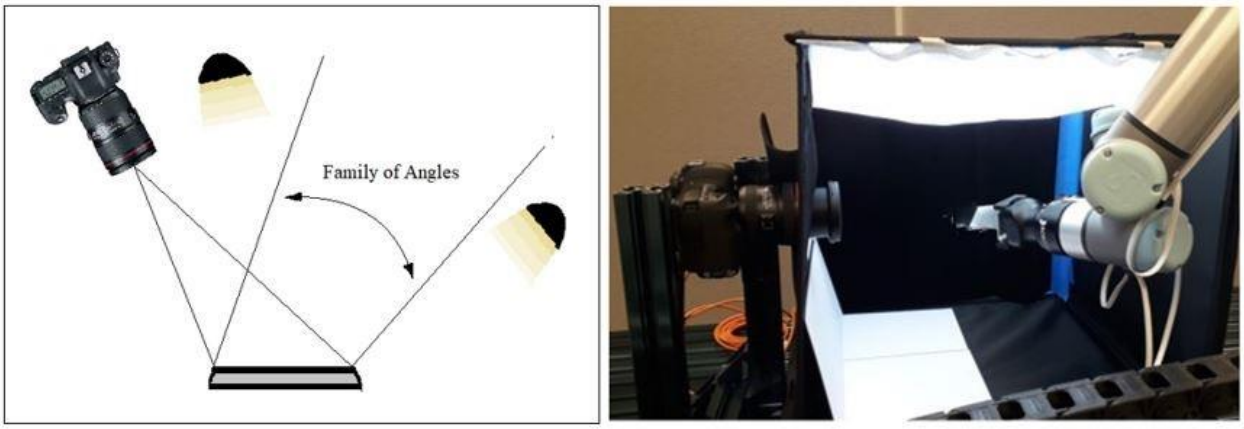

Figure 2.10: Illustration of the "family of angles" criterion (left), and the resulting configuration in the shooting tent (right)

Exposure of an image can be changed by modifying certain parameters of the camera – Aperture, Shutter speed, and ISO. Different situations call for different values of these parameters and varying any one of them changes the amount of light that is incident on the camera sensor; and by

extension, affect the image quality. For example, to capture fast moving objects, the shutter speed can be increased, but it reduces the exposure time resulting in poor illumination. In this application, camera tuning is catered to imaging metallic surfaces and the reasoning is explained as follows:

- Aperture – F/22. A small aperture results in a large depth of field
- Shutter speed – 1/100. As a thumb rule, shutter speed should be inversely proportional to the focal length. Focal length in this application is approximately 70mm.
- ISO – 5000. Controls the sensitivity of the sensor. This value is set qualitatively, by observing the pixel intensity histogram. A well distributed histogram ensures proper exposure.

2.5 Other Elements of the Solution

Given the scope of the project and that it is a collaborative effort with other individuals and organizations, there are certain elements that have been left out of this dissertation. Following is a list, describing briefly, the research questions answered outside of this work:

- Robot motion to pick up the part from the bin – Computer Vision algorithms are used to identify the position of the part to be picked up (within a certain tolerance for error). A hardcoded trajectory is used to remove the part from the foam insert.
- Sophisticated Control of lighting – For better illumination of the part, which maintains the uniformity in imaging between different subregions, a closed loop control of lighting is explored. Since the Shooting Tent has an array of LEDs, selective switching is also a possibility that is being explored.
- The Classifier Network – The scope of this dissertation ends at the compilation of processed images of the surface. The design of architecture of the classifying network is another key aspect of this project which is not covered here.

3. AUTOMATED WAYPOINT GENERATION

3.1 Motivation

Waypoint Recording works well for a complex geometry, with custom-designed components including the gripper, the bit-masking, the corrective methods of feature-based image adjustment, and clever positioning of lighting, Manual. But the procedure has limitations.

- First, it becomes impractical to manually divide a part into smaller segments, if the size of the part is too large. Applying the bit-masking technique in the case of large parts would mean keeping track of hundreds of different binary masks. Because the part can no longer be held by the robot arm, a major change in the setup is required – the camera must be mounted on the arm and the part would remain stationary. Moreover, due to potential errors in camera positioning for different subregions, the bitmasks would not extract the same area reliably. Feature based image transform will be ineffective because the surface being imaged would be contiguous; there will not necessarily be any features to track. Hence, this solution is not scalable with part size.
- Second, different inspection workbenches will differ in dimensions, precision of actuators, and relative positions of components. Each setup will have to be calibrated – that is, a human would have to record waypoints for each part as an overhead step. This becomes a highly subjective inspection pipeline and defeats the original purpose of this endeavor – to remove human subjectivity.

While regulated lighting and a robust classification network might eliminate these potential sources of inconsistencies, a better alternative to circumvent these challenges is needed. Essentially, if the waypoints were derived using geometric properties of the part itself, that would overcome the obstacles of the manual method and would generalize the inspection pipeline for any given part.

3.2 Process

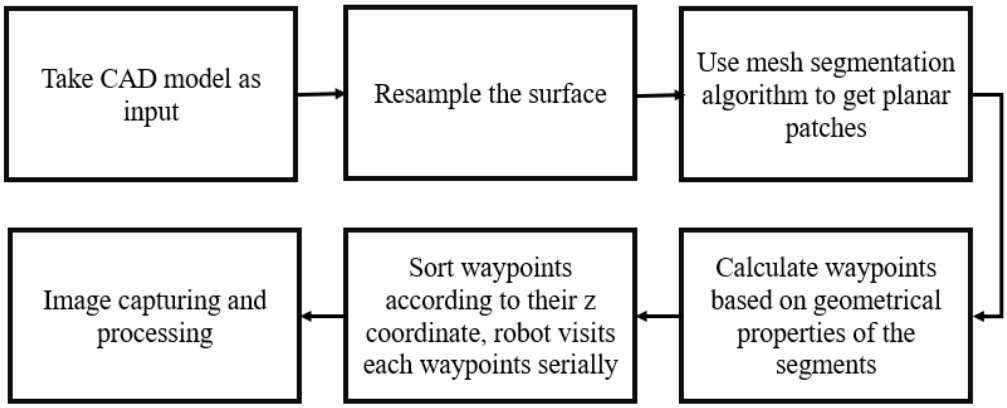

Figure 3.1: Process Flow of Automated Waypoint Generation

The overall procedure is inspired by previous work in the automotive industry, related to inspection of parts either visually or by a Co-ordinate Measuring Machine [10]. Path finding for coverage of 3D object by a robot end-effector is also addressed in more recent work for specific applications, like bush-trimming [11].

There are some important changes in the assumptions made, compared to the manual version of the process and the differences will be mentioned wherever necessary. The size of the part is 1 order of magnitude greater than the camera focal length. As an ideal part for all intents and purposes, a solid cube of length 1 meter can be imagined. The part is assumed to be a convex solid, meaning that its convex hull is geometrically equivalent to the actual shape – although, tests have been conducted to determine the limit of concavity in section 4 of this text. Protruding ‘limbs’, thin sheets and other complicated structures should not be a part of the geometry under consideration. Lighting is assumed to be uniform and conducive to high quality imaging. The CAD model file extension can be anything, if it can be converted to an STL or a PLY file – limited by the capabilities of the Python libraries used for computation.

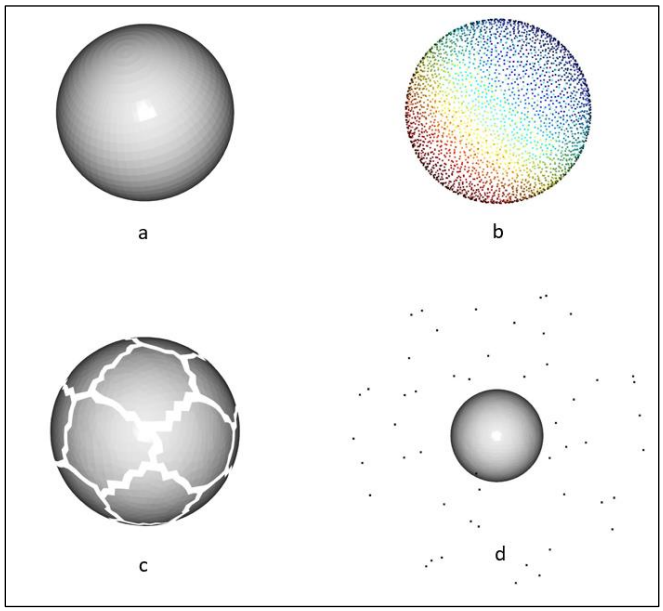

Figure 3.2: Illustration showing the different stages of Automated Waypoint Generation – (a) Input mesh, (b) Resampled mesh, (c) Segmented mesh, (d) Waypoints around the part

The steps to calculate waypoints are briefly explained below, and illustrated in figure 3.2:

1. Input to the pipeline is a CAD model (mesh file), which is a collection of ordered vertices and the different surfaces that are described by them. A resampling of the surface is undertaken for two reasons. First, to eliminate any symmetry in the points for symmetrical objects (Sphere, Cube). Symmetry can be detrimental, because it often implies presence of singular points on the surface as shown. Singular points usually have a high density of triangular surfaces surrounding them, which is an indication of difference in density of points, which adversely affects the subdivision of surface. Second, it is a limitation of the mesh file format in this application that flat surfaces are represented sparsely by points. It is imperative for the segmentation algorithm (step 2) to have a uniform density of points representing the complete surface of the CAD model.
2. The second step is largely a form of patch-based mesh segmentation (explained in section 3.3). Based on the local geometrical properties of the model, its surface is segmented into different regions, and each region is planar within a certain tolerance.

This tolerance is based on the depth of field of the camera. Hence, this step uses the camera model to automatically divide the surface of the part into subregions, as was done before.

3. Now that we have a list of relatively planar regions on the part, each region is divided further into areas that can be imaged at once. That is, each flat region is sub-divided into smaller regions based on the Field of View of the camera. Using conservative values for camera intrinsic parameters, we obtain a list of planar regions, which together cover the entire surface of the part.
4. Finally, using local geometric properties, the position and orientation of the camera for each of the subregions is calculated, such that the patch under consideration will be in focus in the final image. The list of co-ordinates will be sorted based on specific metric calculated using robot joint angles.
5. Following this step 4, with regards to image post-processing, the pipeline is the same, with some necessary changes (explained in section 3.4).

3.3 Mesh Segmentation

Mesh Segmentation is a general problem with applications in many fields like computer graphics, computer aided manufacturing, medical imaging, and others. In this case, it is an important step because of its direct effect on the quality of the image; the depth of field is the constraint that controls the way the part is divided into separate capturable sections. Existing literature suggests that this research field is very well matured, and many solutions, algorithms, and tools are available to subdivide 3D surfaces or meshes.

The algorithms developed for mesh segmentation are based on concepts derived from unsupervised machine learning, image segmentation, computational geometry, and others. Broadly speaking, they can be classified into two types – Part-based Segmentation and Surface-based Segmentation [12]. Part-based is used for separating and identifying different constituents of a mesh, which can be contiguous bodies. Simply put, it tries to mimic the way humans perceive segments in a body. For example, fingers of a hand, legs of an animal. Whereas Surface-based deals with decomposing a given mesh into “patches” and the geometric entities of each patch (vertices, local normal, curvature, etc.) are similar within a certain tolerance. Another way to

classify Mesh Segmentation algorithms is by the strategy adopted, or the broad class of algorithm that is implemented. Some examples are – Region Growing, Multi-source Region Growing, Iterative Clustering, Hierarchical Clustering, Spectral Analysis, and others [13].

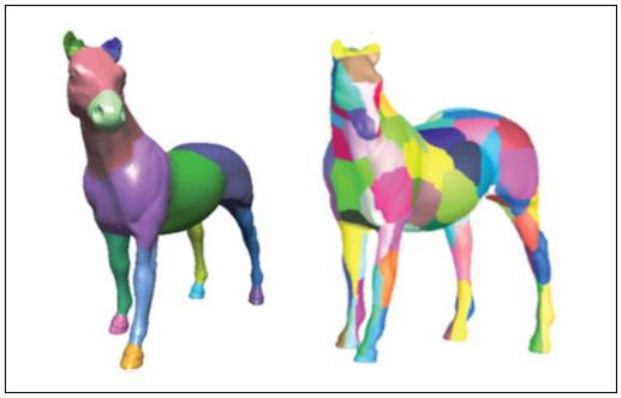

Figure 3.3: Illustration showing the difference between Part-based Segmentation (left), and Patch-based Segmentation (right) [12]

For the inspection problem, Surface-based Segmentation is relevant. Several existing algorithms have the potential to be useful in our case and they were studied thoroughly. One method is described in [14], which applies hierarchical clustering to the mesh, and obtains a list of connected approximately flat subregions. In [15], a hybrid region growing algorithm is implemented and each region is approximated by fitting a primitive 2D shape (ellipse, parallelogram). The result is iteratively improved by minimizing a cost-function which accounts for error in fitting primitives to the derived regions. This algorithm is implemented in code in the CGAL Library written in C++. CGAL also has other useful mesh simplification algorithms, but they require calculating obscure geometrical data and storing it in a customized data structure. Moreover, the functions that are called in code to process the input meshes have parameters that can neither be evaluated qualitatively, nor judged beforehand. There are certain algorithms which require a human to select flat surfaces on the mesh as “seed” surfaces for region growing, but that method is counterproductive. In other words, most existing implementations are not catered towards evaluating the result of the segmentation by considering camera constraints. Hence, even if these methods provide good results, it is difficult to know if they produce an optimal solution because

the segments are different subregions for visual inspection. That is why there is a need for a custom segmentation algorithm, which works well with the constraints that the camera model poses, and the results of which could be evaluated quantitatively.

The segmentation algorithm used in this problem is a combination of Lloyd's algorithm (also known as K-Means Clustering) and Binary Search. It also incorporates the camera model as a factor for determining the size of the resulting clusters. We can ensure that the final segmentation can be evaluated effectively, and the images are guaranteed to be compatible with the classifier standards. The steps of the algorithm are explained briefly along with a flowchart (see figure 3.4).

1. The Mesh Segmentation begins following the pre-processing step of resampling the mesh surface. The mesh is converted to an equivalent pointcloud which has the same points in space and is ordered in the same way.
2. The origin of the local coordinate system of the pointcloud is moved to its center of gravity, Local normal vectors associated with each point are calculated. Each normal is computed as the average of its neighboring normal vectors.
3. For each point in the pointcloud, we now have its coordinates x_i, y_i, z_i, and the direction of its normal u_i, v_i, w_i, where u_i, v_i, w_i are the components of the normal along the x, y, z axes respectively. x_i, y_i, z_i are then normalized so that each coordinate belongs in the closed interval [-1, 1]. This is called feature normalization [16]. We then construct a vector [x_i y_i z_i u_i v_i w_i] for each point in the pointcloud and stack them to form a p×6 Feature Matrix, where p is the total number of points.
4. The next step is to apply K-means Clustering algorithm on the Feature Matrix and obtain clusters of points, where each cluster is a collection of points that are geometrically similar – that is, they lie close to one another and share the same local normal.
5. An important parameter to be supplied to the K-means algorithm is k (number of clusters). Since it is impossible to know k for any given mesh, we determine it iteratively using the Binary Search algorithm. We start with an initial guess for k based on the total number of points p, and adjust its value by evaluating the resulting clusters – we fit a bounding box for each cluster and determine if the largest bounding box (hence, the most curved cluster) is capable of being inscribed in the depth of field of the camera. In this manner, the binary search is wrapped around the K-means method (see figure 3.4).

The output of this procedure is a group of lists of point indices, where each list is a cluster. While it is ensured that each cluster's curvature is within bounds of the depth of field, the 2D area of the cluster can be greater than the field of view of the camera.

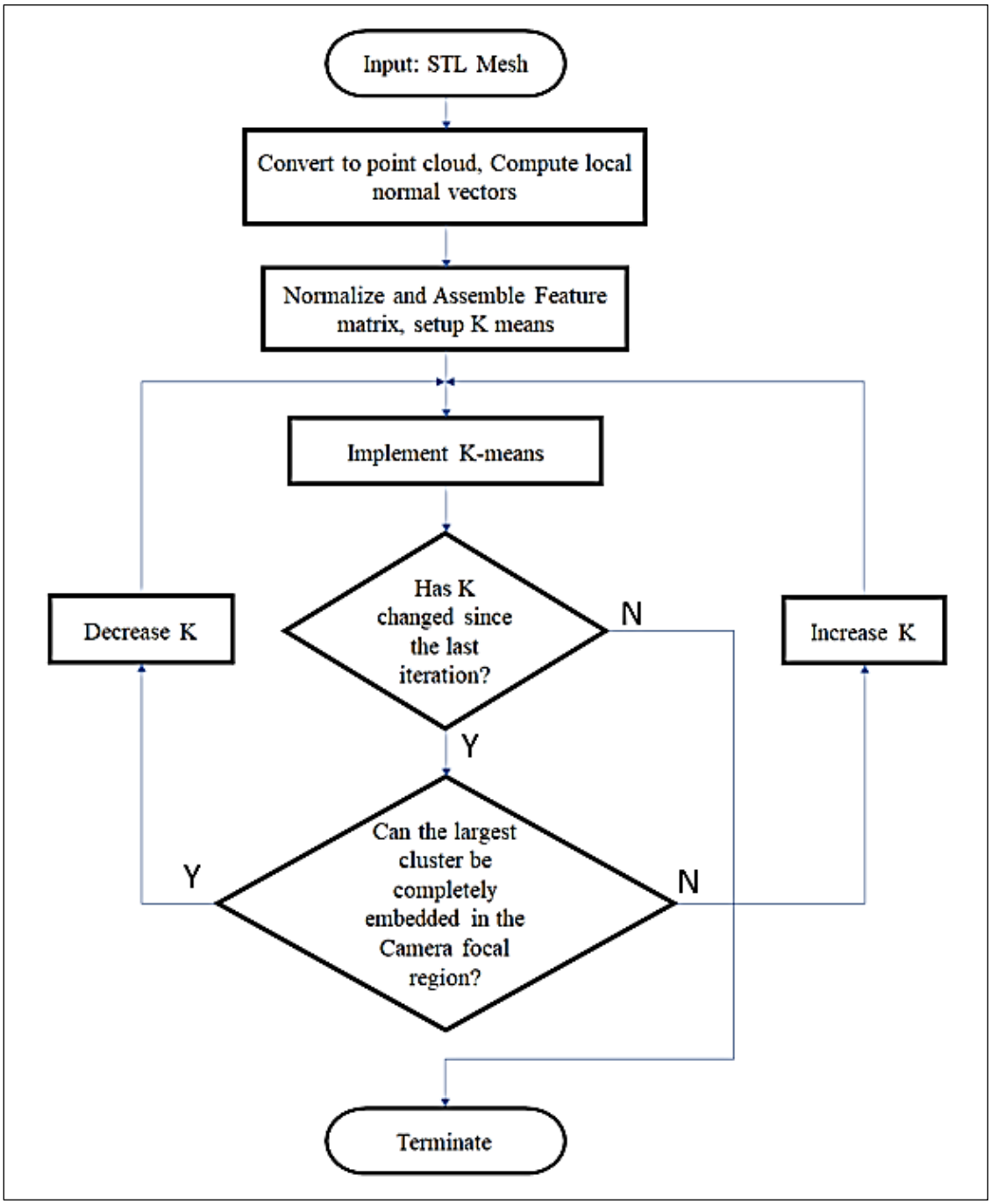

Figure 3.4: Flowchart of the Custom Mesh Segmentation Algorithm

3.4 Waypoint Generation and Optimal Trajectory

As seen in the previous section, it is possible that there exist some clusters which span more than the imaging area of the camera. These clusters need to be further divided into subregions that will be the final areas that will be imaged in one shutter trigger.

It is established that the patches obtained by the current version of the K-Means based algorithm can be encompassed by the depth of field. If some changes are made to the decision criterion for setting *k*, the imaging area constraint can also be incorporated, and each patch can be imaged in one shutter trigger. More specifically, instead of checking whether the bounding box depth is larger in one of its dimensions, we check whether the bounding box volume is comfortably larger than the volume of the cluster in question.

Once we have the final list of patches that will be the subregions, we can calculate the actual waypoints that the robot end-effector must visit. The best quality image is obtained if the subject lies within camera focus, and this is possible if the surface being captured is perpendicular to the optical axis and at a distance away from the camera equal to the focal length. To translate this requirement, the mean normal of all points belonging to one cluster is calculated, and the 3D coordinate of the waypoint associated with that cluster is obtained by simply translating a fixed length along that normal. This fixed length is indeed the focal length of the camera. The missing piece is the direction of the camera optical axis – that is simply the opposite of the mean normal.

After the waypoints are found, the robot needs to visit them optimally – unplanned trajectories will cause redundant motions, and cost time and energy. This can be classified into what is called the "Coverage Path Planning" problem. Literature review suggests that there are a lot of factors that come into play when planning robot trajectory, and consequently there are many ways to approach this problem. 3D cellular decomposition, random sampling-based coverage, grid-based planning are some popular examples [17]. The simplest method would be to treat this as a 3D rasterization problem and sort the points with respect to their positions on one of the axes. In this case, the Z-axis is chosen, and the waypoints are sorted according to their *z* coordinate. This is equivalent to slicing the mesh by various XY planes with different *z* coordinates, and serially visiting points that lie approximately on those planes.

An important aspect of this inspection solution is the bit-masking technique. The automated waypoint generation produces a list of the position and orientation of each point the robot needs to visit. Any sophisticated 6 degree of freedom robot arm has the capability to take coordinates in space and solve the forward kinematics problem to achieve that state for its end-effector. In our case, the UR5e with a camera attached to it can track the coordinates that the algorithm produces as output. However, depending on the actual coordinates, the final configuration of the robot can have one degree of freedom and still satisfy the imaging constraints. For example, the "Wrist 3" joint of the UR5e might still be able to rotate while the object remains in focus. Because the Field of View is rectangular, this can result in inconsistent imaging for separate parts, and potentially result in partial coverage of the surface. To solve this, the bit-mask shape is modified to a circle, since all images are symmetric about the optical axis of the camera, removing all ambiguity in the final image – all captures will be independent of the robot orientation. The circle diameter is set to be slightly less than the smaller field of view dimension.

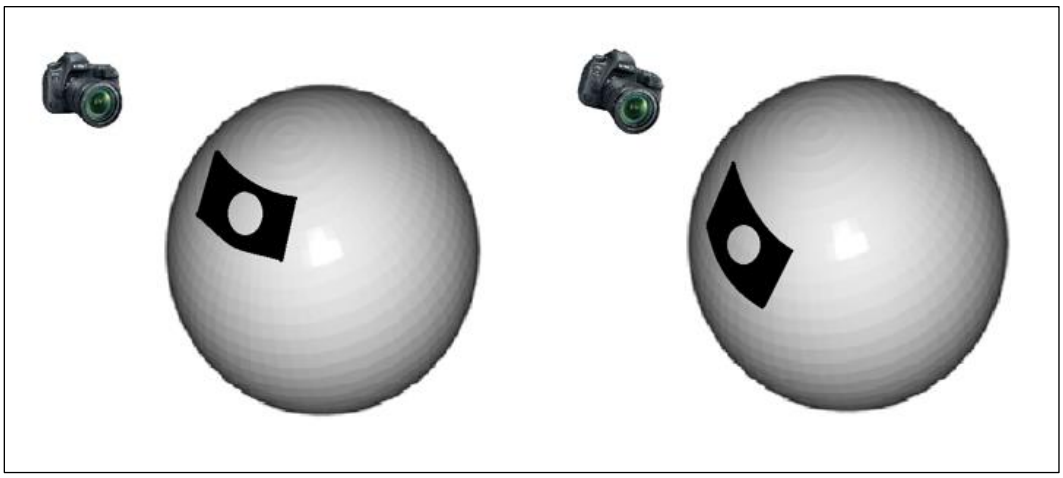

Figure 3.5: Illustration of the advantage of modified bit-mask. Although both waypoints are the same but with different orientation of the camera, circular bitmasks extract the same area

4. EXPERIMENTS AND SIMULATION

4.1 Setup for Experiments and Simulation

The experimental setup for the Manual Waypoint Recording has been explained in chapter 2. Since the assumptions made for the Automated Waypoint Generation problem simplify the problem, the experiments conducted for validation are purely in software. The script for the algorithm is written entirely in Python. For simulation of the robot arm, Gazebo, an open source physics engine is used. It can be easily integrated with ROS, the open source middleware for robotics. For the motion planning part, MoveIt, an open source motion planning framework is used. For computations, Open3D, a 3D data processing library is used. It handles mesh files by using its native data structure called TriangleMesh and has modules capable of geometric data manipulation. For the K-means clustering, the Scikit-Learn library is used. SolidWorks is used for designing and modeling test parts, and obtaining STL files. The camera model used for geometrical considerations is largely a pinhole camera model. Values for depth of field, Field of View for the set camera parameters are calculated using [18] and [19]. The ROS environment is illustrated in figure 4.1. The end effector has a camera attached to it, which emulates the actual sensor – we can observe the viewpoint of the camera as shown in figure 4.2. The robot cannot visit every waypoint in the segmentation algorithm output because some of them will cause collision with the ground. That is why, a feature is built in the code, which allows us to selectively filter out certain waypoints. The outcome is illustrated in figure 4.3.

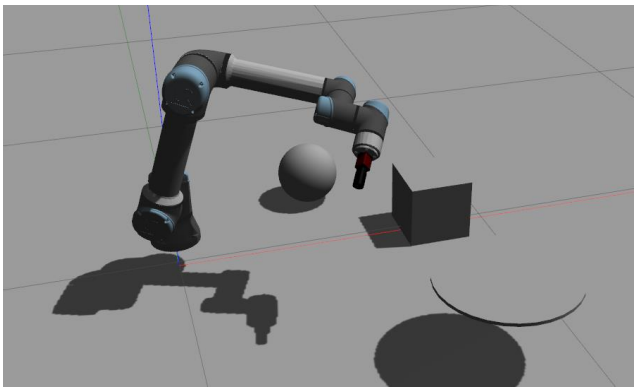

Figure 4.1: The Gazebo and ROS simulation environment

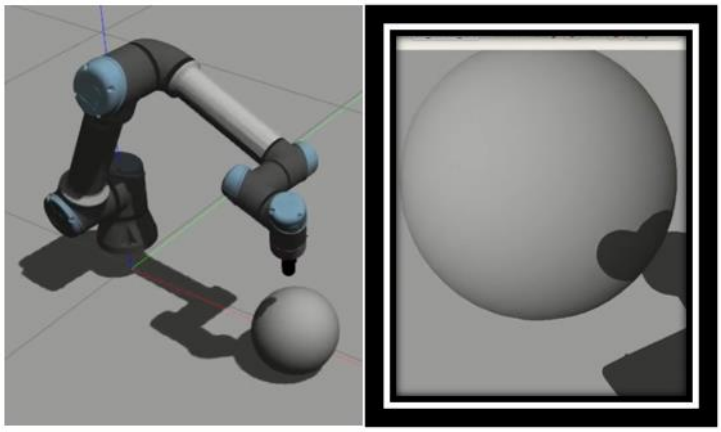

Figure 4.2: Robot at a waypoint of a sphere (left), View of the camera plugin used (right)

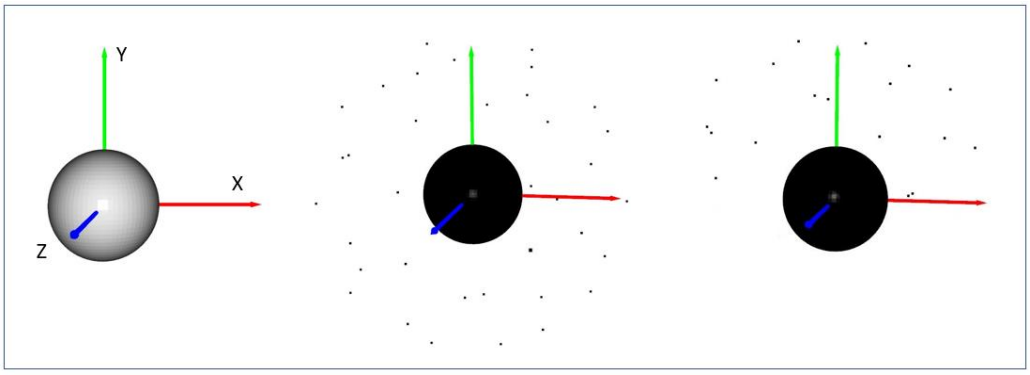

Figure 4.3: Illustration showing the waypoint filtering feature – (left) Input mesh, (center) All computed waypoints, (right) Waypoints of -*y* axis filtered out

4.2 Results for Manual Waypoint Recording

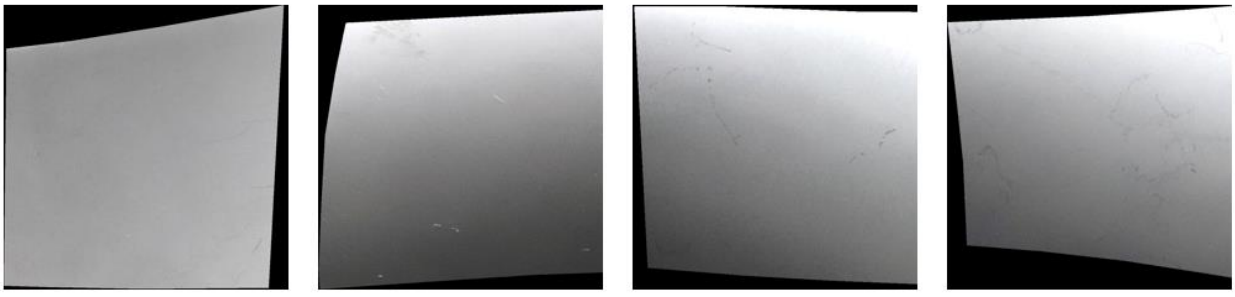

Figure 4.4: Final images of some subregions from the Manual Waypoint Generation process

Qualitative analysis and empirical evidence from colleagues suggest that the classifier can identify at least 90% of the defects, which is an excellent result in terms of image quality, and points to the success of the Manual Waypoint Recording process. To estimate the efficacy of the solution, the numbers reported by experts at an aerospace manufacturing company were consulted. They predict that it would take about 5 months to break even the cost of human inspectors if the current version of the solution were implemented in limited capacity. It is also pointed out that the average cost of inspection per part would be at most $1/6^{th}$ of the cost of human inspectors. Please see Appendix for detailed figures.

4.3 Mesh Segmentation Algorithm for different Topologies

A qualitative analysis of the segmentation algorithm was conducted to test its capabilities on various geometries. For this purpose, CAD models of test shapes were created in SolidWorks. The surface area of all of these meshes were within 10% of 3000000 square millimeters, to maintain uniformity. Certain constants in the code were adjusted if the clustering failed – number of points needed for resampling, and the initial value of clusters (k). The camera parameters were constant for all parts, and the values were as follows:

- Focal length = 70 mm
- Horizontal field of view = 28.8°
- Vertical field of view = 19.5°
- Distance of subject = 300 mm
- Image Area = 154 mm × 102 mm
- Bitmask Area = 6400 square mm
- Depth of Field = 20 mm

The results are depicted in the following illustrations. As expected, the algorithm is robust for more convex geometries (A, B, C). For thin geometries (D, E) the parts need to be resampled with dramatically high number of points. Finally, parts with “limbs” (F, G) need to be resampled similar to thin parts, but they also result in clusters that are not “round”. In other words, imaging with those clusters will lead to background being included. We can conclude that given enough points,

the algorithm works for most geometries – the only disadvantage is the time taken for segmentation.

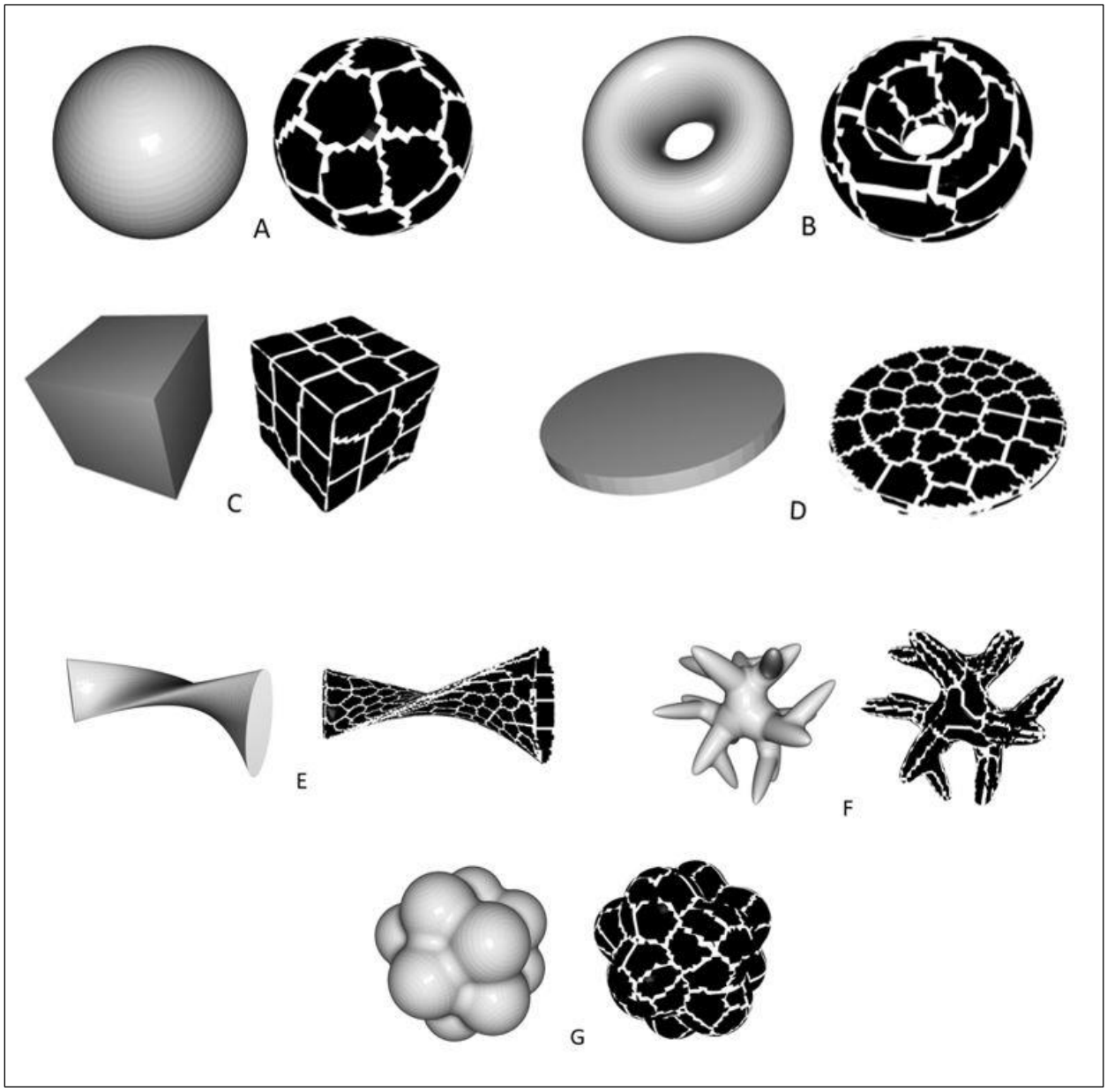

Figure 4.5: Illustration showing qualitative results of mesh segmentation for different geometries

4.4 Results for Thin Geometries

DOF / Thickness	20 mm	30 mm	40 mm
2 mm	88000	88100	79000
3 mm	24700	24900	25200
4 mm	7700	19000	8600
5 mm	6100	7600	7600
6 mm	7100	5700	7100
7 mm	6300	6400	6300
8 mm	6100	6100	6200

Table 4.1: Minimum number of points required for resampling of thin parts for a successful segmentation

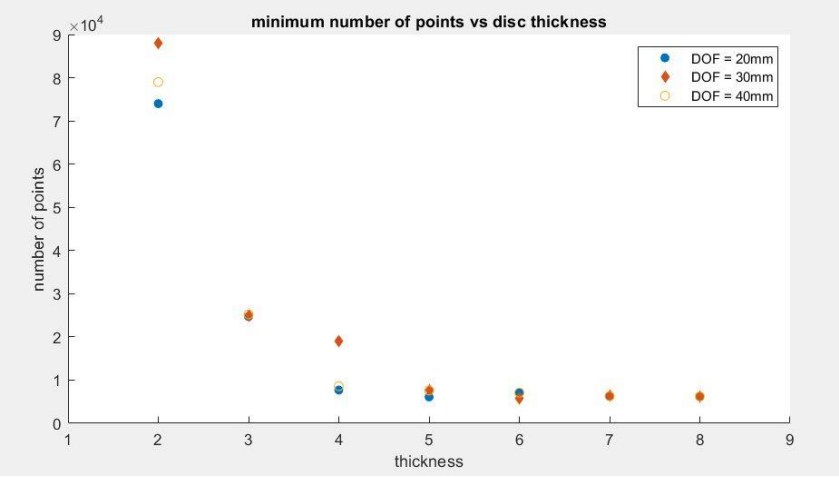

Figure 4.6: A plot of the minimum number of points required for resampling of thin parts

As seen in section 4.2, the number of points required for the algorithm to be successful depends heavily on the part "depth". In this experiment, we determine the relationship between part thickness and the number of points. For this purpose, a disc of variable depth is chosen – the segmentation will be implemented on successively thinner discs, and in each instance, the

approximate minimum number of points for which the clustering was successful was recorded. This experiment is carried out for 3 different depth of fields of the camera. Results are presented in a table and illustrated as well.

Evidently, thinner parts need significantly more points for the resampling stage. The ratio of surface area to volume for any given part is a good indicator of its thickness and can be a more suitable parameter to determine the mathematical relationship – it can be argued that figure 4.7 is a linear or quadratic relationship, which is intuitive. The reason behind this trend lies in the way the algorithm checks for optimal number of clusters. For thin parts, the surface area per volume is very high – which means a greater number of clusters are required but number of points available are sparse. This leads to the failure of the segmentation algorithm because the bounding boxes for bounds-checking do not have enough points to be formed. Therefore, more points are needed.

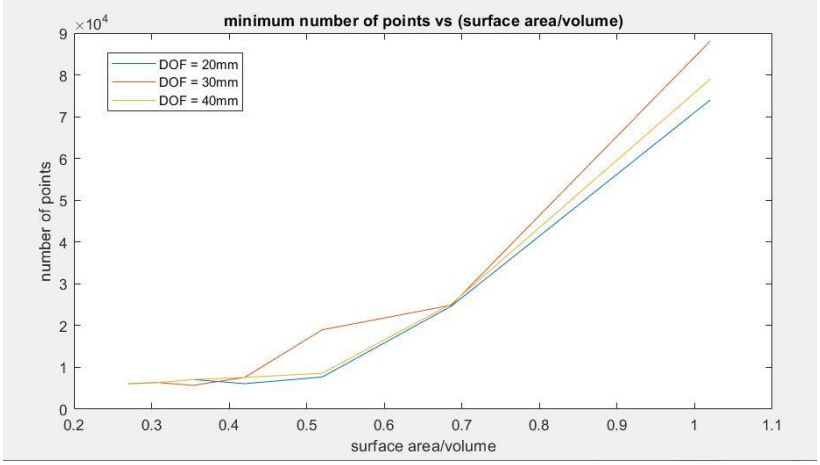

Figure 4.7: A plot of the minimum number of points required for resampling of thin parts, with respect to a better metric (surface area per volume)

4.5 Time Complexity

To evaluate the time required for the algorithm, separate instances of segmentation were run on spheres of the same size, but with different number of points. K-means algorithm in common library implementations are efficient. This efficiency is reflected in the results illustrated in the figure. This mesh segmentation algorithm has linear time complexity, or *O(n).*

Points	Time
4000	12.0
5400	21.7
6100	27.2
7200	29.5
8800	35.9
10500	53.2
12100	59.4
13200	84.8
15800	105.4
38600	286.9
63000	523.5

Table 4.2: Time required for segmentation of given number of points

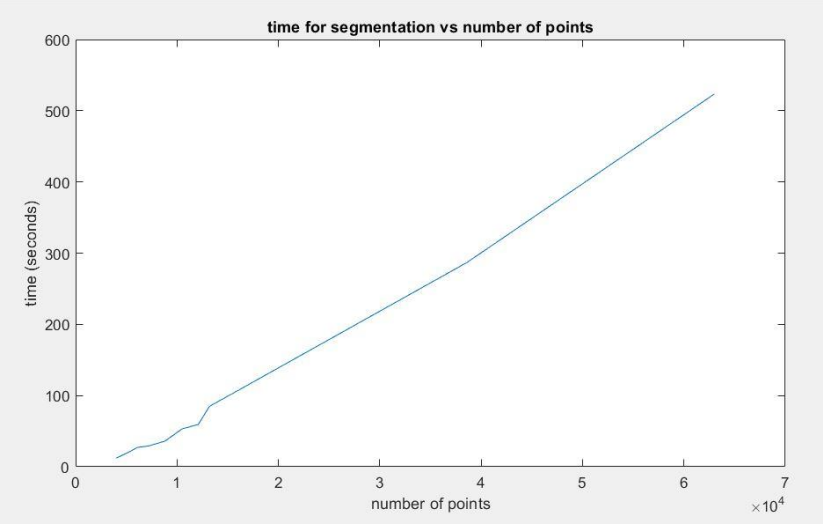

Figure 4.8: A plot showing the linear time complexity of the algorithm

5. LIMITATIONS AND FUTURE WORK

5.1 Limitations

Even though the field of Mesh Segmentation is mature and there exist a lot of solutions for different applications, when we consider the general surface inspection problem, it is difficult to arrive at a universal solution. This problem can be compared to the automated spray-painting problem – where the goal is to derive waypoints and a path for optimal paint layer covering of any given CAD model [20]. Both these problems are also limited by the same factors – there are countless possibilities for requirements of a generic surface inspection, and in all cases, any algorithm should be suitably modified to suit the geometrical constraints posed by the subject of inspection.

With regards to this algorithm, there are a few known limitations, and they were addressed in the scope of the problem in section 1.4. K-Means is a non-deterministic algorithm – applying this process to the same object multiple times would result in different segments. This means that it is also possible that the algorithm will settle in a local minimum in terms of the final clusters. To circumvent this, as part of the procedure this segmentation is applied to any new part in question until a satisfactory clustering is obtained, and these waypoints are reused for all subsequent parts of the same geometry.

Another limitation which was insinuated earlier in this text, is that it works on convex geometries. To clarify, the algorithm itself works well to an extent, but the rest of the procedure which involves using the same bitmasks for each capture limits the applicability to parts with sharp changes in surface geometry. For example, the algorithm will identify six faces of a cube, but the circular bit-masks will result in the final images to depict background in certain cases.

Finally, the trajectory to visit each point as derived in this text is by no means the optimal path. Many assumptions are implicitly made to simplify the problem and not exceed the scope of this work; for example, trajectories being collision free, part being entirely within the robot workspace, robot can localize the part.

5.2 Future Work

After the success of the Manual version of this solution, the Automated version was the natural next step towards a more generalized process. The process described in here is a combination of unsupervised machine learning and computational geometry. Even if these respective fields are mature, the inspection problem is too complex and subjective to external parameters like part geometry, lighting quality, among others. There is scope for improvement in many elements of the algorithm.

For instance, the main limitation is that we are restricted to apply it to convex and solid geometries. There is potential to extend the capabilities of this algorithm to parts with thin features. Another enhancement would have been to make an interactive GUI to select specific surfaces for inspection. Furthermore, for situations where we obtain flat patches that span a vast area, better rasterization paths can be planned to avoid overlap between different captures. Additionally, the waypoints can be traversed optimally by implementing sophisticated graph algorithms. A good place to start would be to treat this as a Traveling Salesman Problem in 3 dimensions. It can also be argued that an optimal path can be calculated by minimizing a cost function which considers parameters like energy expenditure by the robot, robot joint positions, among others. Finally, we can apply an improved version of this algorithm on bigger parts and have a multi-robot coordinated inspection cell or a mobile robot with a manipulator to cover the large surface area.

BIBLIOGRAPHY

[1] Campbell, F. "Inspection Methods—Overview and Comparison." *Inspection of Metals—Understanding the Basics* (2013): 1-20.

[2] Newman, Timothy S., and Anil K. Jain. "A survey of automated visual inspection." *Computer vision and image understanding* 61.2 (1995): 231-262.

[3] Bonnin-Pascual, Francisco, and Alberto Ortiz. "On the use of robots and vision technologies for the inspection of vessels: A survey on recent advances." *Ocean Engineering* 190 (2019): 106420.

[4] Siegel, Mel, and Priyan Gunatilake. "Remote enhanced visual inspection of aircraft by a mobile robot." *Proc. of the 1998 IEEE Workshop on Emerging Technologies, Intelligent Measurement and Virtual Systems for Instrumentation and Measurement (ETIMVIS'98)*. 1998.

[5] Neogi, Nirbhar, Dusmanta K. Mohanta, and Pranab K. Dutta. "Review of vision-based steel surface inspection systems." *EURASIP Journal on Image and Video Processing* 2014.1 (2014): 50.

[6] Wu, W. Y., and C. C. Hou. "Automated metal surface inspection through machine vision." *The Imaging Science Journal* 51.2 (2003): 79-88.

[7] Xue-Wu, Zhang, et al. "A vision inspection system for the surface defects of strongly reflected metal based on multi-class SVM." *Expert Systems with Applications* 38.5 (2011): 5930-5939.

[8] Johnsen, Sönke. *The optics of life: a biologist's guide to light in nature*. Princeton University Press, 2012.

[9] Martin, Daryl. "A practical guide to machine vision lighting." *Midwest Sales and Support Manager, Adv Illum2007* (2007): 1-3.

[10] Sheng, Weihua, et al. "Automated CAD-guided automobile part dimensional inspection." *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*. Vol. 2. IEEE, 2000.

[11] Kaljaca, Dejan, Bastiaan Vroegindeweij, and Eldert van Henten. "Coverage trajectory planning for a bush trimming robot arm." *Journal of Field Robotics* 37.2 (2020): 283-308.

[12] Rodrigues, Rui SV, José FM Morgado, and Abel JP Gomes. "Part-based mesh segmentation: a survey." *Computer Graphics Forum*. Vol. 37. No. 6. 2018.

[13] Shamir, Ariel. "A survey on mesh segmentation techniques." *Computer graphics forum*. Vol. 27. No. 6. Oxford, UK: Blackwell Publishing Ltd, 2008.

[14] Garland, Michael, Andrew Willmott, and Paul S. Heckbert. "Hierarchical face clustering on polygonal surfaces." *Proceedings of the 2001 symposium on Interactive 3D graphics*. 2001.

[15] Cohen-Steiner, David, Pierre Alliez, and Mathieu Desbrun. "Variational shape approximation." *ACM SIGGRAPH 2004 Papers*. 2004. 905-914.

[16] Shanker, Murali, Michael Y. Hu, and Ming S. Hung. "Effect of data standardization on neural network training." *Omega* 24.4 (1996): 385-397.

[17] Galceran, Enric, and Marc Carreras. "A survey on coverage path planning for robotics." *Robotics and Autonomous systems* 61.12 (2013): 1258-1276.

[18] https://www.photopills.com/calculators

[19] https://www.scantips.com/lights/fieldofview.html

[20] Chen, Heping, Thomas Fuhlbrigge, and Xiongzi Li. "Automated industrial robot path planning for spray painting process: a review." *2008 IEEE International Conference on Automation Science and Engineering*. IEEE, 2008.

Appendix A:

Python code for Automated Waypoint Generation

```
"""
CODE
This is the final version of the entire pipeline
"""


# ____________________________________________________
# IMPORTS
# ____________________________________________________

import numpy as np
import open3d as o3d # . . . . . . . . . . . . . . . . Open3D
from sklearn.cluster import KMeans # . . . . . . . . . K-means
from sklearn import preprocessing
from sklearn.preprocessing import minmax_scale



# ____________________________________________________
# CONSTANTS
# ____________________________________________________

part_num = 1 # . . . . . . . . . . . . . . . . . . . . object
h_FOV = 150 # . . . . . . . . . . . . . . . . . . horizontal field of view (mm)
v_FOV = 100 # . . . . . . . . . . . . . . . . . . vertical field of view (mm)
camera_area = 3.14*(v_FOV/2)**2 # . . . . . . . . . circular area (mm^2)
bbcheck_scale = 100.0 # . . . . . . . . . . . . constant used in K-means
cam_dof = 20 # . . . . . . . . . . . . . . . . . . depth of field (mm)
Ry = (np.array([[1, 0, 0], [0, 0, 1], [0, -1, 0]]))
Rx = (np.array([[0, 0, 1], [0, 1, 0], [-1, 0, 0]]))



# ____________________________________________________
# DRAW AXES
# - A  function that draws axes to help with waypoint filtering
# ____________________________________________________

def get_axes():
    x_axis = o3d.geometry.TriangleMesh.create_arrow(cylinder_radius=5.0,
cone_radius=8.0, cylinder_height=400.0, cone_height=30.0, resolution=20,
cylinder_split=4, cone_split=1)
    x_axis.paint_uniform_color((np.array([1, 0, 0])))
    x_axis.rotate(Rx, (np.array([0, 0, 0])))
    y_axis = o3d.geometry.TriangleMesh.create_arrow(cylinder_radius=5.0,
```

```
cone_radius=8.0, cylinder_height=400.0, cone_height=30.0, resolution=20,
cylinder_split=4, cone_split=1)
    y_axis.rotate(Ry, (np.array([0, 0, 0])))
    y_axis.paint_uniform_color((np.array([0, 1, 0])))
    z_axis = o3d.geometry.TriangleMesh.create_arrow(cylinder_radius=5.0,
cone_radius=8.0, cylinder_height=400.0, cone_height=30.0, resolution=20,
cylinder_split=4, cone_split=1)
    z_axis.paint_uniform_color((np.array([0, 0, 1])))
    return x_axis, y_axis, z_axis
x_axis,  y_axis, z_axis = get_axes()



# ____________________________________________________________
# MESH PREPROCESSING
# - Load STL file
# - Translate to its center of mass
# - Resample the surface with suitable number of points
# - Save the modified mesh file
# ____________________________________________________________

# translate
mesh = o3d.io.read_triangle_mesh("mesh/part_{}.STL".format(str(part_num)))
mesh.compute_vertex_normals()
mesh.translate(-mesh.get_center())
o3d.io.write_triangle_mesh("mesh/part_{}.STL".format(str(part_num)), mesh)

# visualize
mesh = o3d.io.read_triangle_mesh("mesh/part_{}.STL".format(str(part_num)))
mesh.remove_duplicated_vertices()
mesh.compute_vertex_normals()
AABB = mesh.get_axis_aligned_bounding_box()
ET = AABB.get_extent() # . . . . . used for waypoint filtering
later
o3d.visualization.draw_geometries([mesh, x_axis, y_axis, z_axis])

# resample and save
num_vertices = int(mesh.get_surface_area()/100) # . . . . . one point for
100mm^2 area
throwaway_pcd =
mesh.sample_points_poisson_disk(number_of_points=num_vertices)
throwaway_pcd.estimate_normals()
mesh, _ =
o3d.geometry.TriangleMesh.create_from_point_cloud_poisson(throwaway_pcd)
o3d.io.write_triangle_mesh("mesh/part_{}_resampled.PLY".format(str(part_num))
, mesh)
mesh =
o3d.io.read_triangle_mesh("mesh/part_{}_resampled.PLY".format(str(part_num)))
```

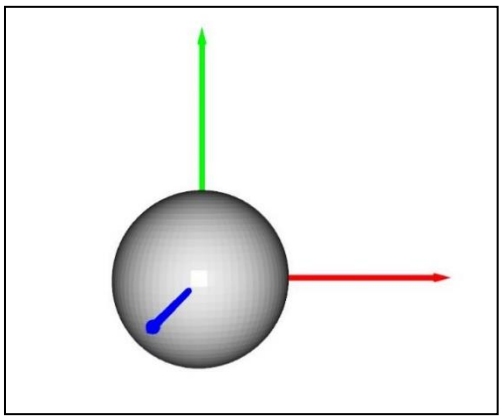

Figure A.1: Mesh with axes

```
# ____________________________________________________________
# DATA COMPILATION
# - Compute vertex locations and local normals
# - Stack all features to make a feature matrix
# ____________________________________________________________

# feature scaling
mesh.compute_vertex_normals()
mesh_normals = np.asarray(mesh.vertex_normals)
mesh_vertices = np.asarray(mesh.vertices)
mesh_vertices = 2 * (minmax_scale(mesh_vertices) - 0.5)

# create feature matrix
loc_w, norm_w = 1, 1 # . . . . . . . . . . weights assigned to location and
normal
data = np.concatenate((loc_w * mesh_vertices, norm_w * mesh_normals), axis=1)



# ____________________________________________________________
# K-MEANS SETUP
# - Custom K means algorithm which includes bounds checking with camera focal
region
# ____________________________________________________________

# KMeans parameters
init_method = "random"
max_iter = 100
num_runs = 10

# kmeans clustering function which uses bounding box as an evaluator
def Kmeans_with_evaluation(init_method, num_clusters, num_runs, max_iter,
data, pcd):
    KM = KMeans(init=init_method, n_clusters=num_clusters, n_init=num_runs,
max_iter=max_iter)
    KM.fit(data)
```

```
    labels = KM.labels_
    cluster_collection = [[] for i in range(num_clusters)]
    for j in range(len(labels)):
        cluster_collection[labels[j]].append(j)
    extents = []
    spans = []
    for i in range(num_clusters):
        temp_pcd = pcd.select_by_index(cluster_collection[i])
        c = temp_pcd.get_center()
        temp_pcd.scale(bbcheck_scale, c)
        temp_pcd_hull, _ = temp_pcd.compute_convex_hull()
        temp_bb = temp_pcd_hull.get_oriented_bounding_box()
        temp_bb_extents = temp_bb.extent
        extents.append(temp_bb_extents[2])
        spans.append(temp_bb_extents[0]*temp_bb_extents[1])
    return (max(extents))/bbcheck_scale,
(max(spans))/(bbcheck_scale*bbcheck_scale)



# ____________________________________________________________
# K-MEANS IMPLEMENTATION WITH BINARY SEARCH
# - Iterative K means clustering of the mesh, with changing K in each
instance
# ____________________________________________________________

# binary search to find optimal k
pcd =
o3d.io.read_point_cloud("mesh/part_{}_resampled.PLY".format(str(part_num)))
bs_high = len(pcd.points)//100
bs_mid = 0
bs_low = 1
while(bs_high > bs_low):
    print(bs_low, bs_mid, bs_high)
    bs_mid = (bs_low + bs_high)//2
    max_depth, max_area = Kmeans_with_evaluation(init_method, bs_mid,
num_runs, max_iter, data, pcd)
    if (max_depth > 0.8*cam_dof) or (max_area > 0.8*camera_area):
        bs_low = bs_mid + 1
    else:
        bs_high = bs_mid
print("optimal no. of clusters: {}".format(bs_high))

# final kmeans
num_clusters = bs_high
KM_final = KMeans(init=init_method, n_clusters=num_clusters, n_init=num_runs,
max_iter=max_iter)
KM_final.fit(data)
labels = KM_final.labels_
```

```
# ____________________________________________________________
# DERIVING DIFFERENT ELEMENTS OF THE MESH
# - Assemble all clusters as a list of meshes
# ____________________________________________________________

# cluster_collection
# list of varible size lists. each nested list is a collection of point
indices that belong to a cluster
cluster_collection = [[] for i in range(num_clusters)]
for j in range(len(labels)):
    cluster_collection[labels[j]].append(j)

# cluster_collection_color
# all clusters colored
cluster_collection_color = []
for i in range(num_clusters):
    temp_pcd = pcd.select_by_index(cluster_collection[i])
    color = np.random.random([3, 1])
    temp_pcd.paint_uniform_color(color)
    cluster_collection_color.append(temp_pcd)
```

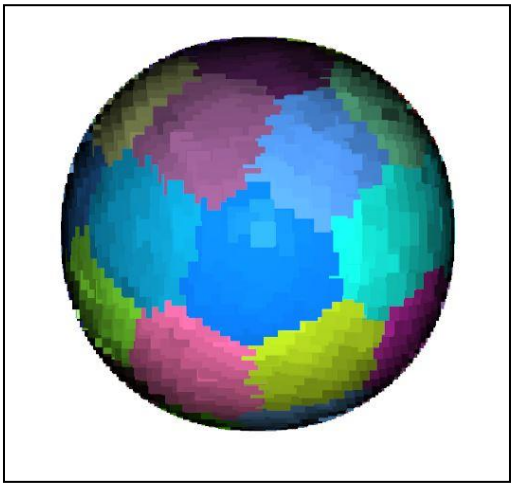

Figure A.2: Segmented mesh

```
# obb_list
# list of all bounding boxes
obb_list = []
for i in range(num_clusters):
    temp_pcd = pcd.select_by_index(cluster_collection[i])
    temp_bb = temp_pcd.get_oriented_bounding_box()
    obb_list.append(temp_bb)
```

```
# ____________________________________________________________
# WAYPOINT GENERATION
# - Find mean location and mean normal vector of each subregion
# - Translate each location along the local normal by a length equal to the
focal length of the camera
# ____________________________________________________________

# mesh_list
# list containting segmented meshes that are final subregions
mesh_list = []
for i in range(num_clusters):
    temp_mesh = mesh.select_by_index(cluster_collection[i])
    mesh_list.append(temp_mesh)
o3d.visualization.draw_geometries(mesh_list, point_show_normal=True)

# waypoints_projection_list, waypoints_normals_list
# lists of coordinate projections and normals of all subregions
waypoints_projection_list = []
waypoints_normals_list = []
for i in range(len(mesh_list)):
    temp_mesh = mesh_list[i]
    temp_mesh.remove_duplicated_vertices()
    temp_mesh.compute_vertex_normals()
    temp_xyz = np.asarray(temp_mesh.vertices)
    temp_uvw = np.asarray(temp_mesh.vertex_normals)
    temp_xyz = np.mean(temp_xyz, axis=0)
    temp_uvw = np.mean(temp_uvw, axis=0)
    temp_uvw /= np.linalg.norm(temp_uvw)
    waypoints_projection_list.append(temp_xyz)
    waypoints_normals_list.append(temp_uvw)

offset = 300
N = len(waypoints_projection_list)
displacements = [offset*norms for norms in waypoints_normals_list]
WAYPOINTS_XYZ_prefilter = [(waypoints_projection_list[i] + displacements[i])
for i in range(N)]
```

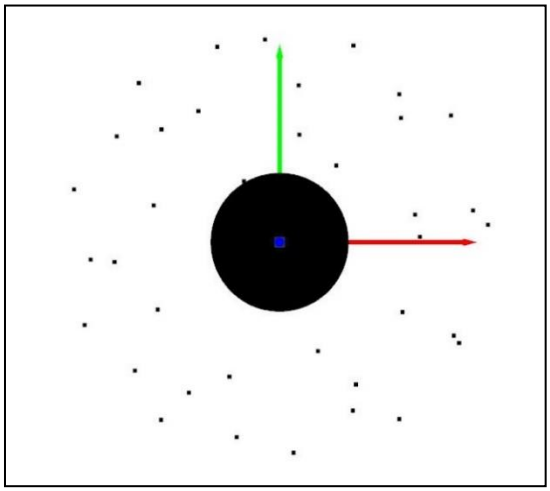

Figure A.3 Waypoints around the mesh

```
# ____________________________________________________________
# WAYPOINT FILTERING
# - Select which axes' waypoints to filter out
# - Fiter out waypoints and assemble new list of waypoints
# ____________________________________________________________

# input axes to reject those waypoints {x, y, z, -x, -y, -z}
reject_list = [0, 0, 0, 0, 0, 0]
extents_for_filter = [np.abs(ET[0]/2), np.abs(ET[1]/2), np.abs(ET[2]/2)]

# filter helper function
def range_checker(point, extents_for_filter, reject_list): # point = nparray
    for i in range(6):
        if (reject_list[i] == 1):
            if (i<3):
                if (point[(i)] > extents_for_filter[(i)]):
                    return False
            else:
                if (point[(i%3)] < -extents_for_filter[(i%3)]):
                    return False
    return True

# filter waypoints
WAYPOINTS_XYZ = []
WAYPOINTS_N = []
for i in range(len(WAYPOINTS_XYZ_prefilter)):
    point = WAYPOINTS_XYZ_prefilter[i]
    allowed = range_checker(point, extents_for_filter, reject_list)
    if (allowed == True):
        WAYPOINTS_XYZ.append(point)
        WAYPOINTS_N.append(waypoints_normals_list[i])
print("no. of waypoints before filtering: {}".format(str(N)))
print("no. of waypoints after filtering: {}".format(str(len(WAYPOINTS_XYZ))))



# ____________________________________________________________
# WAYPOINT VISUALIZATION
# ____________________________________________________________

# convert to pcd and visualize
WAYPOINTS_XYZ_arr = np.array(WAYPOINTS_XYZ)
waypoint_visualization_pcd = o3d.geometry.PointCloud()
waypoint_visualization_pcd.points =
o3d.utility.Vector3dVector(WAYPOINTS_XYZ_arr)
black = np.zeros((3, 1))
waypoint_visualization_pcd.paint_uniform_color(black)
o3d.visualization.draw_geometries([waypoint_visualization_pcd, mesh, x_axis,
y_axis, z_axis])
```

```
# ____________________________________________________________
# WAYPOINT SORTING
# - Sort according to the Z axis coordinate of the waypoints
# ____________________________________________________________

waypoint_queue_unsorted = []
for i in range(len(WAYPOINTS_XYZ)):
    waypoint_queue_unsorted.append((i, WAYPOINTS_XYZ[i][2]))

print(waypoint_queue_unsorted)
waypoint_queue_sorted = sorted(waypoint_queue_unsorted, key=lambda x: x[1])
print(waypoint_queue_sorted)
```

Appendix B:

Python Code for Bitmask Compensation

```
"""
BIT MASK CORRECTION
"""

import cv2
import numpy as np


# ____________________________________________________
# CALIBRATION IMAGE PREPROCESSING
# ____________________________________________________
calibration_image = cv2.imread("extra/calibration_image.JPG")
rows,cols,ch = calibration_image.shape
calibration_image = cv2.cvtColor(calibration_image,cv2.COLOR_BGR2GRAY)
calibration_image = cv2.resize(calibration_image,(int(0.1*cols),
int(0.1*rows)), interpolation = cv2.INTER_CUBIC)


# ____________________________________________________
# SUBJECT IMAGE PREPROCESSING
# ____________________________________________________
subject_image = cv2.imread("extra/subject_image.JPG")
subject_image = cv2.resize(subject_image,(cols, rows), interpolation =
cv2.INTER_CUBIC)
subject_image = cv2.cvtColor(subject_image,cv2.COLOR_BGR2GRAY)
subject_image = cv2.resize(subject_image,(int(0.1*cols), int(0.1*rows)),
interpolation = cv2.INTER_CUBIC)


# ____________________________________________________
# SIFT
# - Feature Detection
# ____________________________________________________
sift = cv2.SIFT_create()
kp_c = sift.detect(calibration_image,None)
kp_c, des_c = sift.compute(calibration_image, kp_c)
kp_s = sift.detect(subject_image,None)
kp_s, des_s = sift.compute(subject_image, kp_s)


# ____________________________________________________
# FLANN
# - Feature Matching
# ____________________________________________________
FLANN_INDEX_KDTREE = 0
index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
search_params = dict(checks = 50)
flann = cv2.FlannBasedMatcher(index_params, search_params)
```

```
matches = flann.knnMatch(des_c,des_s,k=2)
print(len(matches))
good = []
for m,n in matches:
    if m.distance < 0.7*n.distance:
        good.append(m)
print(len(good))


# ____________________________________________________________
# HOMOGRAPHY
# - Find the transform that corrects the subject image with respect to the
calibration image
# ____________________________________________________________
c_pts = np.float32([ kp_c[m.queryIdx].pt for m in good ]).reshape(-1,1,2)
s_pts = np.float32([ kp_s[m.trainIdx].pt for m in good ]).reshape(-1,1,2)
M, mask = cv2.findHomography(s_pts, c_pts, cv2.RANSAC,5.0)
dst_s = cv2.warpPerspective(subject_image, M, (int(0.1*cols), int(0.1*rows)))



# ____________________________________________________________
# DISPLAY RESULTS
# ____________________________________________________________
cv2.imshow("c_img", calibration_image)
cv2.imshow("s_img", subject_image)
cv2.imshow("transformed_s", dst_s)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Appendix C:

Python Code for Camera Control

```
import os
import gphoto2 as gp # package used for controlling camera by USB
import cv2
import numpy as np


class camera_controller:

    # connect to the camera.
    def __init__(self):
        self.camera = gp.Camera()
        self.camera.init()


    # disconnect the camera.
    def __del__(self):
        self.camera.exit()


    # Capture image and save to a path
    def trigger(self):
        # capture image
        temp_img_path = self.camera.capture(gp.GP_CAPTURE_IMAGE)
        # save to temporary storage
        img_path = os.path.join('img_path', temp_img_path.name)
        camera_file = self.camera.file_get(temp_img_path.folder,
temp_img_path.name, gp.GP_FILE_TYPE_NORMAL)
        camera_file.save(img_path)
        return img_path
```

Appendix D:

Quantitative Analysis Report for Manual Waypoint Recording

(as reported by Project Coordinator)

Cost of Manual Inspection by Human Inspectors = $208000 per year

Cost of Manual Inspection by Robots = $72400 per year

Time to break even = 72400/208000 = 4.17 months

	Inspector	Robot
Rate	$135	$37.88
Duty Cycle	80%	95%
Reinspection Rate (MRB)	5%	2%
inspectors/systems	4	2
parts/month	2950	2950
Inspection rate minutes/part	12	6
Cost per part	$27	$4
Amortization Period	5	

Figure D.1: A report depicting the cost benefit of the Manual Waypoint Inspection