Automating box cutting with ROS

Mechanical Engineering bachelor thesis

1st Jesse Bloothoofd
Mechanical Engineering
TU Delft
Delft, the Netherlands
J.S.Bloothoofd@student.tudelft.nl

2nd Roderik Bruins
Mechanical Engineering
TU Delft
Delft, the Netherlands
R.L.W.Bruins@student.tudelft.nl

3rd Rinus Schuijt
Mechanical Engineering
TU Delft
Delft, the Netherlands
M.J.Schuijt@student.tudelft.nl

4th Sebastiaan Timmer
Mechanical Engineering
TU Delft
Delft, the Netherlands
S.J.Timmer@student.tudelft.nl

***Abstract*—This research was commissioned by Ahold Delhaize. The aim was to produce a system using a robotic arm and 3D camera to accurately cut open different sized boxes. The software used for this was ROS (Robotic Operating System). The application Gödel, an open source project designed for scanning and blending surfaces, was used and adapted in order to make it suitable for the assignment. After successful simulations the application was connected to the robot arm ABB IRB1200. After successful merging of the application and robot arm, different knives were tested resulting in the use of a powered circular blade. Lastly, a 3D camera (Asus Xtion) was added to the system to scan the boxes instead of using generated point cloud data. It can be concluded that within a simulation it is possible to automate the box cutting process. When using the 3D camera, small imperfections in the generated point cloud made it hard for the box cutting application to recognise the precise surface. As a result, executing the cut sometimes had a small offset resulting in cutting the cardboard instead of the tape. Due to Gödel being a 2D path planner, the box cutting application can not take small differences in height into account resulting in a part of the box not being cut. Solutions for these issues can be explored by using a more accurate 3D camera and a 3D path planning package. Apart from this, it can be concluded that this project has produced a satisfactory proof of concept.**

I. Introduction

Ahold Delhaize is one of the pioneers when it comes to automating the processes in between the supplier and the customer. Their distribution center in Zaandam has automated almost every single process and it could be only a matter of time before the grocery stores will see fully automated processes too. Robotization will not only speed up certain processes and save costs, it will also take away some tasks that are boring and repetitive for employees and are better suited for a robot.

The company is trying to find out if different stages in logistics can be automated and the process of cutting boxes is one of them. From an engineering point of view this is a challenging task since there is deformation of the material due to stacking and tiny margins in between the box and the products that fill up the box. Simple tasks, such as making a robotic arm move from point A to B are being used for a while now and are widely adopted across different industries. Since technology keeps evolving, 3D cameras can produce point cloud data more accurate and faster than ever before. This creates new opportunities to improve automated industrial processes.

The assignment that Ahold Delhaize gave is to research the options of automating the box cutting process. This research will be done by using ROS, short for Robotic Operating System. ROS is an open source framework for writing robot software. This allows the programmer to build on existing packages. More specific, we will use the Gödel application within ROS. This is a bundle of packages made for point cloud surface detection, path planning and polishing. A brief introduction to both ROS and Gödel will follow in this paper first before it focuses on the actual project.

This paper has the following goals: (1) to describe the problem and the design requirements for the robotic arm, in order to solve it, (2) to give an overview of the steps taken to achieve the goal (a detailed description of this can be found in Appendix 1), (3) to provide an overview of the results, (4) to discuss and clarify these results and (5) to provide a starting point for further development in this field.

This design study has been conducted by bachelor Mechanical Engineering students of the TU Delft's Mechanical Engineering department as a graduation project.

1) Introduction to ROS

According to the wiki of ROS, ROS is an open source set of software libraries and tools (”About ROS,” n.d.). Open source means that there is information and existing code on the internet that can be reused. After discussing the main goals with the project supervisors, the choice was made to use a pre-programmed application within ROS called ”Gödel”. This will be used as a starting point for designing a box cutting robot. Gödel is an application using ROS that can be used for locating and blending (polishing) surfaces.

ROS makes it easy to use and combine packages. Packages can contain one or multiple nodes. Nodes are files of code that fulfill a certain purpose (”Nodes,” n.d.). Communication between nodes is done by topics (”Topics,” n.d.). An

application is usually built from multiple packages, nodes and topics work together and each have different purposes for the application.

2) Introduction to Gödel

Gödel is an application using ROS that is part of the open-source Scan-N-Plan™ project of ROS Industrial (”Scan-N-Plan™”, n.d.). On their website, ROS-Industrial describes itself as follows: ”ROS-Industrial is an open-source project that extends the advanced capabilities of ROS software to manufacturing.” (”ROS-Industrial”, n.d.) Gödel is an application that consists of a collection of packages. It is divided into two clearly distinctive processes. The first process is the scan process which contains the scanning of the operating space of the robot. The robotic arm first moves to a specified scan pose. When the robotic arm is positioned at the scan pose, a scan of the operating space is made. The second process is the plan process which contains the detection of all surfaces in the obtained point cloud of the operating space, generating a blending and laser scanning path and making a plan how to actuate the robot arm.

Gödel is an adequate option to start with, since the blending and laser scanning movements are comparable to box cutting movements. Therefore, Gödel is used for this design study.

The original Gödel application is meant to first make a scan of the operating space. A point cloud is a 3D cluster of dimensionless points that are represented as dots. Together these dots approach the shape of objects. A point cloud can be obtained either by generating a fictive point cloud or by using a 3D-camera which generates a point cloud.

After scanning the operating space and obtaining a point cloud, Gödel detects all flat surfaces of the point cloud within the operating space. Gödel then plans two paths over each detected surface: one path for the blending tool (red trajectory) and one path for the laser scanner (yellow trajectory) *(figure 1)*. There are various end effectors attached to the robot arm that are used for the blending application, for instance a Keyence laser scanner and a blending tool. These however are not useful for the box cutting application.

Gödel can be used to control a real robot arm when a connection is established. The program constructs both a simulation and an execution of the blending processes and the laser scanning processes. The simulation can be used to inspect the motion of the robot arm before executing it. Executing makes the robot arm actually move. If the robot arm makes unexpected movements during the simulation, the operator can choose not to execute the motion.

The default robot arms included in the Gödel application are from ABB. However, if a different arm is required due to specification requirements such as minimum operating range, the specific model can be put into the Gödel application by making some changes to the support files and the MoveIt configuration files.

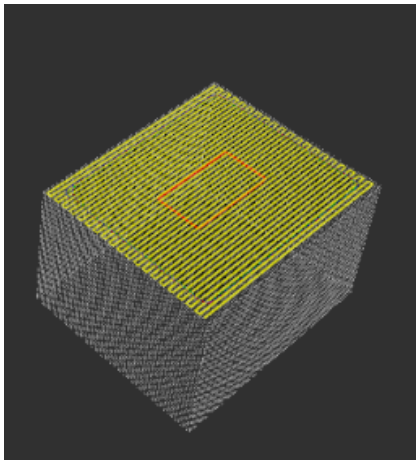

Fig. 1. Example of a blending path (red) and a laser scanner path (yellow), produced by Gödel

Most of the files in Gödel are written in the c++ programming language. Next to that, Gödel makes use of .yaml files and some .STL files. The c++ written files are the main code, the .yaml usually contain parameters and the .STL files are files that contain the physical objects which could be used as a visual and collision model independently. RViz, a visualisation tool within ROS, is used to visualize the robot and the operating space in a 3D space.

II. Objectives and requirements

The main objective of Ahold Delhaize is to explore the possibilities of automation in their processes. One of these processes is cutting boxes open. An essential aspect that comes with that is convincing the managing board whether cutting open a box using a robot is possible in the first place. This implies that it is important to be able to show a demonstration of a proof of concept, rather than having a completely worked out idea but nothing to demonstrate. What this means is that concessions can be made in order to get the product working. The main requirements are the ability to locate the box, to recognize the top surface of the box and to determine the correct path necessary to cut it open.

A. Requirements

There are three requirements for the design study.

- ROS and Gödel have to be used for this project.
- The robotic arm should be able to cut different sized boxes.
- The box cutting application should be able to recognize surfaces independent of location and angle.

B. Objectives

This project consists of three main objectives. To achieve these objectives, sub-objectives were set as a guideline for the project.

Main objective 1: Getting a working simulation of a robot arm cutting open a box in the Gödel application.

Sub-objectives

- Determine the best way to cut open a box
- Get a box point cloud into Gödel
- Insert the cutting tool as end effector
- Alter the laser scan path planning
- Remove path planning process for blending
- Select the top surface of the box
- Execute the scanning
- Give the cutting tool the correct orientation
- Selection of an appropriate robotic arm which is also available for testing

Main objective 2: Transitioning from a simulation in the Gödel application to controlling a real robot with the same movements.

Sub-objectives

- Establish connection with the ROS server on the robotic arm
- Design an end effector for box cutting
- Build an end effector for box cutting
- Get the conveyor belt working

Main objective 3: Implementation of a 3D camera in the Gödel application to scan real boxes.

Sub-objectives

- Select an appropriate 3D camera and install the correct drivers
- Set the correct scan pose
- Let the box cutting application make a 3D snapshot
- Reset parameters for surface detection
- Solve path planning deviation due to imperfect point cloud

III. Methods

A. Main objective 1: Simulation

The method for achieving the first main objective was going through files, getting an overview of Gödel and looking for specific parts of code that could be changed in such a way that the objectives would be met.

To get a better understanding of how Gödel is structured, a tree structure was made of how all launch files and nodes are related. This is done by opening each launch file and going through them. By excluding one node or launch file at a time, it could be determined which nodes and launch files are of importance and which ones are not. By applying this method, it was possible to narrow down the number of files to look through.

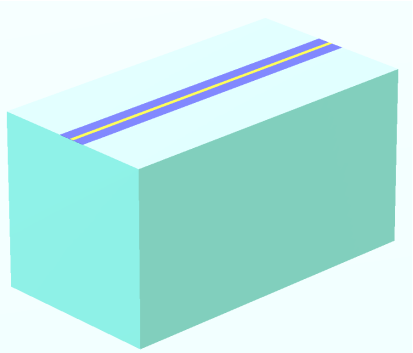

Fig. 2. Example of a box; Indigo represents the tape on the box, yellow the areas to be cut

1) Determine the best way to cut open a box

First, an analysis of the design request has been made. The following three methods to cut open a box were considered:
1. Cut the box open through the middle of the top surface of the box. With this approach, two assumptions are made. The first assumption is that the gap and tape will always be exactly in the middle of the top surface of the box. The second assumption is that the box only has tape on the top surface and not any tape at the side surfaces.*(figure 2)*
2. Cut the box open through the middle of the top surface and along the two shortest edges. With this approach, the same assumption is made that the gap and tape will always be exactly in the middle of the top surface of the box *(figure 3)*.
3. Cut the box open by cutting off the whole top *(figure 4)*.

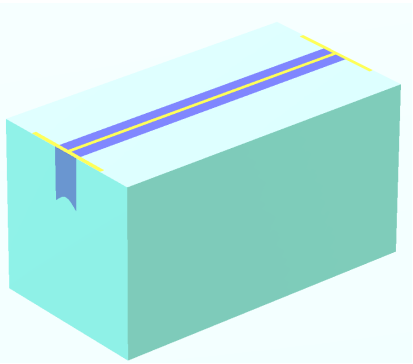

Fig. 3. Example of a box; Indigo represents the tape on the box, yellow the areas to be cut

Out of these three options, the first one appeared to be the most achievable within the time span of the design study. Primarily since Gödel its path planning seemed compatible

with the path that we needed for this option. The advantage of this approach is that the most important part, which is the tape along the length, can be cut with one simple movement.

The second option can be divided into two parts: 1, cutting the tape along the length like the first option and 2, cutting the tape on the sides. A single cut through the middle is not enough to open most boxes because both lids will still be held in place by the tape on the sides. This approach would be able to open most boxes on the market, however the path planning is different from Gödel its standard paths, so it had to be written from scratch. Because the two parts can be separated, even if the more complex movement of cutting the edges does not work according to plan, the box cutting robot can still cut open the tape in the center *(figure 3)*.

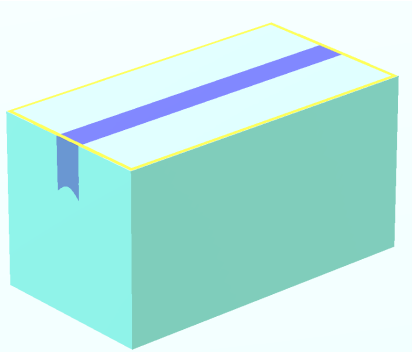

Fig. 4. Example of a box; Indigo represents the tape on the box, yellow the areas to be cut

The third option *(figure 4)* required the end effector to go through the cardboard completely, whereas the first two only required cutting the tape. Taking the preservation of the products in the box into consideration, the first two options seemed better. When the tape has to be cut, the thickness of the cardboard functions as a cutting margin. Having to cut through the first layer of cardboard eliminates this partly and thus increases the risk of cutting the products inside the box.

2) Get a box point cloud in Gödel

Since Gödel originally has three separate horizontal flat point clouds, this needed to be changed. The three surfaces were replaced by a point cloud of a box. In order to accomplish this, the height of one of the surfaces was increased and the width and length were set to the dimensions of a standard box. The other two surfaces were removed. This resulted in a point cloud of a single box.

3) Insert the cutting tool as end effector

A .STL file of the new end effector had to be produced in order to replace the original end effector. To get this done, a model of the cutting tool was made in Solidworks. This file was then converted into a .STL file and implemented into Gödel. The box cutting application needs this .STL file in order to prevent a collision with itself. To not slow down the simulation the .STL file was built from basic figures. It consists only of a few squares and cylinders which represent the basic shape of the end effector.

All parts of code that would call upon the original Gödel end effector were deleted except for the blending tool link that was directly attached to the robot arm. In the code for this part, some modifications were made so that it would call upon the .STL file of the newly created knife. Also the knifes orientation compared to the old end effector had to be changed.

4) Alter the laser scan path planning

Gödel generates paths for the laser scanner and the blending tool. The blending path that is generated is projected along the edges of a surface with a slight offset and the laser scan path makes zigzag movements (called 'slices') over the surface. For this designs study, it seemed more relevant to use the laser scanner path and alter the number of slices to just one. This resulted in one slice exactly through the middle along the length of the top surface (*figure 5*).

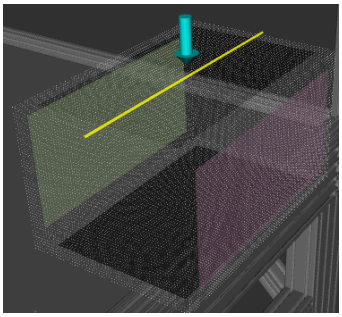

Fig. 5. Path planning when only cutting the tape in the middle

After the cutting tool completed cutting through the middle of the top surface of the box, it also has to cut the tape along the two shortest edges of the box. For that purpose, the end effector has to make a 90 degrees turn over the z-axis. To prevent any damage to the cutting tool, the robotic arm first moves 5 centimeters above the top surface before making this rotation. Also the translation between the two shortest edges is completed 5 centimeters above the top surface for the same reason. It would make more sense if the box cutting robot first cuts one of the two shortest edges, then cuts through the middle and finishes it by cutting the other shortest edge. However, this is not possible since the robotic arm does not turn the 90 degrees quickly enough within the short distance it would travel. The system performs all actions overall in a more rapid way now it only has to perform one 90 degree rotation over the z-axis.

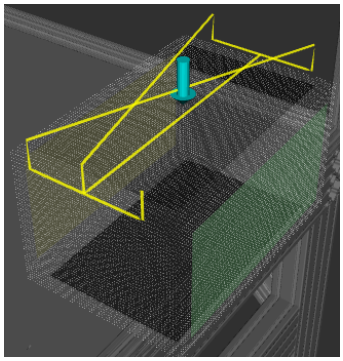

Fig. 6. Path planning including the edges

5) Remove path planning process for blending

In the original Gödel codes, a blending path and a laser scan path are created for each surface. Since it was decided to use the laser scanning path generation for the box cutting robot, the blending path generation is ignored.

6) Select the top part of the box

Another part of the path planning is solely selecting the top surface of the box. In the original Gödel application, the operator has to manually select the surfaces of interest. For the box cutting application, only the top surface is of interest. In order to automate the process of cutting open boxes, the selection of the top surface should be done automatically. The solution for this was making the box cutting application select the surface from the point cloud with the most dots. Most cardboard boxes have a rectangular shape where the sides with the largest surface area and therefore the most dots, are the top and the bottom one. From these two surfaces the top one is automatically selected.

7) Execute the scanning

While running Gödel, the simulation in Rviz performs both a simulation and an execution for the blending. However for the scanning part Gödel only showed a simulation. This was observed because the simulation is a transparent version of the robot in RViz and the real one is not transparent. The transparent simulation did perform the scanning, the other one did not. Since the scanning part is the part that is used, a way had to be found to also perform the execution. Otherwise, when trying to use Gödel on a real robot arm, the robot arm would never execute the movement and only a simulation in Rviz would be run. In the original file, Gödel would look for a connection with the Keyence laser scanner. However in the box cutting application, the Keyence laser scanner is not used. In order to make the script work regardless, all the references to the Keyence laser scanner in this file were deleted.

8) Give the cutting tool the correct orientation

After replacing the original end effector for the cutting tool, the cutting tool moved parallel to the surface. To cut properly the cutting tool has to be orientated perpendicular to the surface. This unwanted movement happened because in Gödel the laser scanner is mounted at the end effector with a different relative location and orientation. The points in the yellow trajectory *(figures 5 and 6)* contain not only information about the location, but also information about the how the robotic arm has to be oriented towards the trajectory. By altering this orientation, the cutting tool will always move perpendicular to the surface.

9) Selection of an appropriate robotic arm which is also available for testing

The default robotic arm used in Gödel is the ABB IRB2400. Because this robotic arm is not available at the Mechanical Engineering Department of TU Delft and the second objective is to implement Gödel to a real robotic arm, it was necessary to search for a substitute. Since the Yaskawa GP25 robotic arm is actually available in the Mechanical Engineering Department, it was chosen to substitute the IRB2400. Another advantage of this arm is that it has a larger reach and therefore could reach each side of a box with ease.

Time was invested to implement the Yaskawa GP25 in Gödel and getting it working correctly. Due to lack of space and lack of available drivers for the actual Yaskawa GP25, it was decided to look for other robotic arms available. It was decided to use the ABB IRB1200 robotic arm since this model was available at Robohouse (Delft, Netherlands). Though being a small robotic arm, it still met the necessary requirements to use for the box cutting application. Since it was already implemented in Gödel it was available for use instantly.

If preferred, other robotic arms can be implemented in Gödel. It could be convenient for any further research and development to use a substitute robot arm. Instructions for implementing a different robotic arm can be found in Appendix 2.

B. Main objective 2: Real robot arm

After getting the simulation in Rviz to work, it was time to move on to the actual ABB IRB1200 at RoboHouse.

1) Establish connection with the ROS server on the robotic arm

In order to make connection with the ABB IRB1200, a tutorial was followed. This tutorial was written by a TU Delft employee and specifically written for the ABB IRB1200 at Robohouse. The tutorial is not available to the public. After following the steps of this tutorial (this includes installing the correct drivers) and setting the right network settings, connection with the ABB IRB1200 was established. In addition, some changes were made to the launch file to make

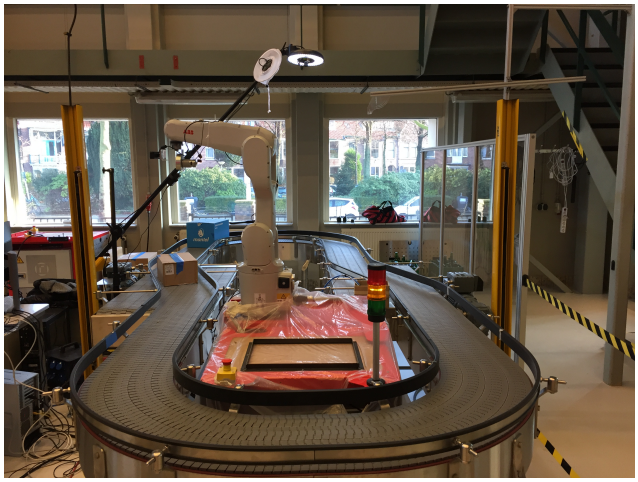

Fig. 7. Setup and operating space of the ABB IRB1200 used in this project

Fig. 8. Utility knife end effector

Fig. 9. Pizza cutter end effector

it work with the box cutting application.

2) Design an end effector for box cutting

Three different kind of cutting tools were considered to use as end effectors in order to cut open the boxes. They were all mountable on the same flange which makes it easy to quickly swap between them. All three end effectors have their own set of advantages and disadvantages.

The first type of cutting tool was a utility knife *(figure 8)*. This is the cheapest and most durable option, however it is potentially dangerous and might have problems with initiating the cut. Typically static friction of this type of knife is high while dynamic friction is low resulting in unwanted forces on the robot arm and cardboard box.

Secondly, a circular blade was considered. In this case a pizza cutter was used *(figure 9)*. Since this blade is rotating it does not have the tendency to push the box out of its way when faced with resistance. Another advantage is that it is not that sharp, making it safer to mount on a robot arm. The lack of sharpness however also requires this blade to be pushed down in the box harder which might deform it. It was also possible to fix the circular blade to see if it was more successful in cutting boxes open.

The third option was a circular blade which was powered by a DC-motor from a hand drill *(figure 10)*. Since this blade is always rotating, there is no static friction therefore there will not be massive differences in friction and the box is less likely to move when cut. Disadvantages would be that this is the most dangerous end effector and since it involves moving parts it is the least durable. The end effector was designed in such a way that the smooth blade of the pizza cutter could be mounted on the motor, as well as two other blades with different sized teeth.

Fig. 10. End effector (circular blade + 3D scanner) mounted at the end of the ABB IRB1200 robotic arm

CAD models of all three end effectors were made in Solidworks in order to easily implement them in the Gödel application.

3) Build an end effector for box cutting

The end effectors were all constructed with the help of a lathe and a milling machine. All were constructed with tolerances of less than 0.1 mm in order not to deviate from the CAD-models, therefore reducing the risk of inconsistencies between simulation and the real world.

4) Get conveyor belt working

The last sub-objective was to automatically feed new boxes to the robot arm without getting dangerously close to it. Thankfully a conveyor belt enclosed the robot arm at the location *(figure 7)*. The belt is easily controlled via the same controller as the one used to manually move the robot arm. Since this was a last minute addition it was only manually controlled.

C. Main objective 3: 3D camera

Up to this point, cutting boxes with the robotic arm and the box cutting application has no potential for opening boxes on an industrial scale. Since the system uses a fixed point cloud, the location, size and orientation of the real box would have to be exactly the same as the one in the simulation. The dimensions of the point cloud in the box cutting application

could be altered in order to cut different sized boxes, but this would still work far from ideal. According to the contact person of Albert Heijn, the company has around 26,000 different sized boxes. A library could be made containing all the different sized boxes, but then the system would have to be manually changed every time. Moreover it would still not allow for damaged boxes.

To cut open boxes on an industrial scale, the process of determining the size, location and orientation of the boxes should be automated with the help of a 3D camera.

1) Select an appropriate 3D camera and install the correct drivers

Due to the high costs of new 3D cameras, only 3D cameras available at the Mechanical Engineering Department of TU Delft were considered for this purpose. There were two Ensenso cameras and one Asus Xtion available at the department. The Ensenso cameras in general have a better resolution so it was the preferred option. However, due to a lack of available ROS drivers for the Ensenso cameras they could not be used. In contrast, there were already ROS drivers available for the Asus Xtion camera (library openni2). Therefore, the decision was made to use this 3D camera. This 3D camera is mounted at the end effector.

2) Set the correct scan pose

In order to correctly translate and rotate the generated point cloud into the world frame, the scan always has to be made from a pre-set location and orientation. This location and orientation is called the scan pose. Since the 3D camera is mounted at the end effector, the robotic arm has to move towards this scan pose before making a scan of the operating space. The scan pose is set at location (0.55;0.0;0.8) (in meters) with respect to the robotic arm and with an orientation in the negative z-axis direction.

3) Let the box cutting application make a 3D snapshot

After the robotic arm moved to the scan pose, a snapshot of the operating space is made. This is done by saving the point cloud generated by the Asus Xtion at that exact moment. In the original Gödel application, a point cloud of a perfect box was placed into the world frame. This point cloud was then used by the surface detection of the Gödel application. In the box cutting application, instead of implementing a point cloud of perfect box the snapshot of a point cloud generated by the Asus Xtion is placed into the world frame. The camera has to be calibrated so this point cloud is placed and transformed into the world frame in such a way that the origin of the point cloud has a location and orientation that corresponds with the location and orientation of the real 3D camera mounted on the robotic arm. After this, the point cloud is used by the surface detection of the box cutting application *(figures 11 and 12)*.

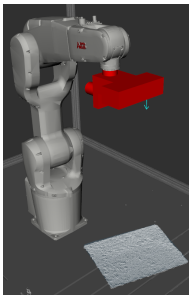

Fig. 11. 3D scan without path

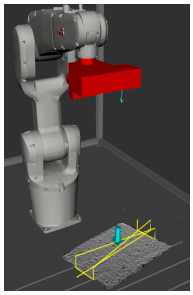

Fig. 12. 3D scan with path

4) Reset parameters for surface detection

A consequence of using imperfect point clouds generated by the Asus Xtion is that the box cutting application was not always able to detect all surfaces. To fix this problem, some parameters used to determine what points form a surface were changed. By altering these values, the box cutting application is able to sufficiently recognize surfaces in the imperfect point cloud. These new values were determined experimentally until the surface detection gave satisfactory results.

5) Solve path planning deviation due to imperfect point cloud

Due to the imperfect point cloud, the generated laser scan path had a small rotation regarding to the surface (Appendix 1 figure 1). This was solved by determining the angle of this faulty rotation and give the laser scan path the same angle of rotating in negative direction (Appendix 1 figure 2).

IV. Results and Observations

A. Main objective 1: Gödel simulation

When putting a fictive box point cloud in the simulation, the box cutting application was able to only select the top part of the box and draw one straight line across its length, as well as a line across both edges. After this the simulation and the execution of the cutting movement was performed correctly in Rviz. This all worked with the ABB IRB 1200 robot arm.

B. Main objective 2: ABB IRB 1200

1) Getting Gödel to work with the robot

First a connection was established with the ROS server on the ABB IRB 1200. After that Gödel was made compatible with the arm. When launching the Gödel application the real robot would now follow the exact movements as shown in Rviz when moving to the scan position and executing the planned path. When a box with the same dimensions as the

point cloud was placed on exactly the same location in the real world, the robot arm would perform a movement exactly across the middle where its planned path was.

2) The end effector

There were difficulties with the box cutting performance of the real robot arm. First of all there was a problem with the utility knife. The static friction of the knife was so great that the box was malformed before it started cutting the cardboard. This resulted in a cut that was not very clean. It had the tendency to push away boxes that were too light. This result however was anticipated and the prime reason for the construction of the circular blade. The circular knife however was not that sharp and had trouble getting through the tape at all.

The solution for this was the powered circular blade. This terminated all static friction. First the smooth blade of the previous circular knife was tested. This setup created a very clean cut in the tape but it was still not optimal. Some boxes that were tested had a very small gap between the cardboard at the top that the tape covered. In that case the knife had to cut some of the cardboard, for which this smooth blade was not ideal. It created so much friction that there was a risk of setting the box on fire. The next step was to use a circular toothed blade. This resulted in a broader cut eliminating the need for very high accuracy. Lastly the size of the teeth of the blade was increased in order to try to cut through the material faster. The blade with larger teeth however seemed to be less succesful than the blade with smaller teeth.

C. Main objective 3: Asus Xtion 3D camera and autonomous box-cutting

1) Asus Xtion 3D Camera

The surface of the point cloud data obtained from the Asus Xtion 3D Camera has reasonable accuracy and surfaces were detected when the 3D camera was positioned properly above the box. The 3D camera did have problems creating point clouds of shiny surfaces. The edges of the scanned boxes were a bit less clear and less defined than the middle section. The box cutting application was also able to detect the environment around the scanned boxes. If there were smooth surfaces close enough to the box, there is also the option to select these as a detected surface.

2) Path planning and execution with the Asus Xtion 3D camera camera

Once the surface was detected the box cutting application would project a straight path through the middle. A small mistake could be observed once or twice when the box rotated a bit underneath the camera. Overall the path planning was accurate and a correct path was calculated and executed. A box can be seen cut open all the way through *(figure 13)*.

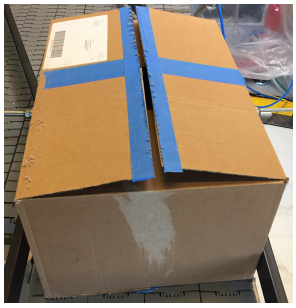

Fig. 13. Good and clean cut through the end

Fig. 14. Failure halfway through

When the box would have a dent or any imperfection that would make the top of the box not completely flat, it showed that the blade would not cut into the box deep enough or not at all at some points leaving it closed, which can be observed in *(figure 14)*. The cut only went only halfway through in the Z-axis, leaving one part of the box only scratched instead of cut.

V. Discussion and Recommendations

1) Box point cloud and path planning

The results show that with the right modifications, Gödel was able to perform all the required tasks in the simulation that were listed in the sub objectives. The recognition of the correct part of the box, followed by a path planning, simulation and execution all worked smoothly.

2) ABB IRB 1200

After connecting with the ABB IRB 1200 and making Gödel compatible with it, the arm performed as expected. As long as the box was placed at exactly the same location as in the simulation, the robot arm would always make a straight precise cut along the path that had to be followed. One of the main objectives of this project was making the robot arm able to perform the task correctly, no matter the location, size and orientation of the box, as long as it was in the robot arm its range. This worked out well.

3) The Asus Xtion 3D camera

The camera worked well enough when only making a cut through the middle of the box. Either the accuracy or the calibration fell short when trying to include cutting the tape on the sides of the boxes as well. It appeared the sides of top surface of the box were not always scanned with the required accuracy needed in order to plan the paths on the sides correctly. Higher camera accuracy will result in higher accuracy cuts and would also be useful if the box cutting application was converted to 3D path planning. This option was already available for the Gödel application but at the time of writing it was still experimental. The current box cutting

application only uses 2D path planning. Therefore it was not that important to detect small bumps and imperfections on the surface of the box. The box cutting application its surface detection makes an average estimate of the entire surface. This cancels out small imperfections and might change the average surface height a little in the world frame. Therefore, even if these bumps were detected with precision, the current box cutting application would not follow the bump properly.

A limitation of the camera was the minimum distance required with respect to the surfaces that needed to be detected. According to the descriptions of the product this is about 0.8m but in reality it still worked correctly to about 0.6m. If the distance is less than 0.6m, empty spots will appear in the point cloud of the box which results in bad path planning. Moreover, the box needs to be accurately placed under the 3D camera because its scanning width is not great. When the box is placed partly outside of the camera its scan width, a part of the scanned surface will be missing which also results in bad path planning.

4) The cut

The offset in the cut that sometimes showed when cutting open the boxes could be explained either by imperfections in the box confusing the path planning, or imperfections in the 3D scan. When the surface scan has edges that are rough, it is likely that this will have an effect on the path planning. Imperfections in the box will also turn into imperfections in the point cloud, which could pose the same problem. Though, when the offset caused the blade to go through the cardboard instead of the tape it would still work because the blade is able to go through cardboard without any problem. Cutting through this cardboard should be done cautiously. When the products in the box are tightly packed inside the box, they are likely to touch top lid of the box. In this case, deformations of the box can induce a path that goes partly through the products and damage them.

5) The end effector

It was not the main focus of this design study to design the most efficient end effector. The one used in the project had to be manually turned on with a button each time the robot is about to be cut. This requires a person to turn it on which in essence does not make the robot autonomous. To make the robot autonomous it is possible to keep the blade spinning all the time. However for safety reasons this might not be the best idea. When the blade is spinning it also creates vibrations which might affect the 3D camera. Ideally, the end effector should be implemented in the box cutting application and be programmed to start each time the robot starts the cut.

6) Safety

When working with a robotic arm, or any machine in general, safety should always be taken into account. In the current setup of the project there is a virtual wall which stops the robot arm its movements if you trespass it. Next to this virtual barrier there is also a physical barrier consisting of a Tensabarrier and marking tape on the floor. Lastly, when the robot arm is actuated, there is always someone that has hands on a kill switch to turn the robot of in case of an emergency or an unexpected movement.

The end effector most used was a custom made cutting tool powered by a dc motor connected to a battery. There are several ways to stop the spinning blade if necessary. Either the power cord can be pulled out of the battery or the button that turns it on and of can be switched. When working on the robot, the power cord is always pulled out so it is not possible that the blade spins up unexpectedly. Even though there is two ways to turn the blade off, there probably should be an extra safety measure. For instance an emergency brake on the blade itself. Making sure the blade also turns off when trespassing the virtual wall or pressing any emergency stop would be a good solution. It should both be connected to the ROS server and to the electrical circuit that is connected to the virtual wall for this to work properly. For the unlikely scenario the blade breaks while spinning, there is a aluminium casing around the blade that should prevent shards from flying in all directions. Though there are no calculations done to the energy of the flying shards and the maximum capability of the aluminium to catch these shards. Instead, an intuitive overdimensioning approach has been taken. For personal safety, safety goggles were worn by people around the robot arm when the cutting tool was spinning.

7) The effective speed

Replacing a human with a robot must have a solid reason. Whether this is speed, cutting costs or for safety reasons. Speed and cutting costs can go hand in hand, and for Ahold Delhaize this is one of the reasons they wanted this design study to be performed. At this point there can be made improvements on speed, which in turn will also save costs because more products can be opened in the same time.

The path planning at this point is the process that up the most time. This can probably solved with a better kinematic solver. This is the program that makes the calculations needed to plan a path. Another way in which this can be solved is by raw power e.g. a faster computing chip that performs the calculations.

A different way to speed up the entire process is to make the robot cut boxes while the conveyor belt still moves. This is a bit tricky to implement in the box cutting application at this point, because the 3D camera is on top of the robotic arm. This gives issues because the robotic arm would have to move above the box to detect and therefore move with the same speed as the conveyor belt. A possible solution would be to split the scanning and the cutting. For example, the scanning could be done at a point on the conveyor belt before

the robotic arm performs operations.

8) Keeping the boxes stationary

The conveyor belt that is located around the ABB IRB1200 robotic arm is made from a material with a smooth surface. This offers little to no grip. For boxes with a heavy content this is not a big issue, but for lightweight boxes this can cause the box to slide over the conveyor belt when the blade pushes against it.

To solve this problem during testing the boxes were filled with heavy concrete blocks to make sure that they wouldn't move. In a more realistic scenario boxes can be a lot lighter then the ones used for testing. A possible solution to keep the boxes in place is to use a material with a high coefficient of friction for the conveyor belt links. Even with this there is still a chance certain boxes will slide over the conveyor belt, and maybe a totally different approach that secures boxes into place should be implemented like locking them into place.

9) Other solutions/approaches to the boxcutting problem

While a robotic arm gives a universal solution to opening boxes thanks to its versatility, it lacks speed and efficiency. Time is lost because the 3D camera is attached to the robotic arm. To scan, the robotic arm has to return to its scan pose every time it has cut a box because there is a minimal scanning height. This in combination with inertia affects the robot arm its effective working speed. Thus limiting the amount of boxes handled in a certain time frame.

One solution that has been discussed early in the project is a conveyor belt with boxes on it and a 3D camera on the side that just measures the height of the boxes. A horizontal band saw could quickly adjust to the height of each of the incoming boxes and cut off the top sheet. This increases overall speed, but with damaged boxes decreases accuracy since this cut will always be linear. Also there is a bigger chance that products get damaged if they are close to the top lid of the box.

The knife used in this case was spinning with several thousand revolutions per minute. The teethed blade then cuts the cardboard in very fine pieces, creating dust. In many environments this dust will not be acceptable and it should therefore be extracted from the blade. Another approach could be a slower spinning higher torque engine on a smooth blade. Both options would result in a better system.

Lastly there was a small problem with the path planning. The way this path planning was performed was by scanning the top surface of the box and planning a path in the middle of the longer side. This worked fine until a square box was scanned. The path planning would not always plan the path over the tape, but sometimes it planned its path perpendicular over the cardboard. This would be solved if the software had some kind of detection tool for the tape.

VI. Conclusions

The aim of this project was a pioneering exploration of the possibilities of automation in the cutting of boxes. The goals were to see if a proof of concept for this application could be made, for both a simulation, and a real world application that would cut open an actual box, using a fictive point-cloud. The third goal was diving into the possibilities in 3D scanning in order to fully automate the process.

Inspecting the results, it can be concluded that within a simulation it is possible to automate the box cutting process. The path planning and execution in the simulation worked perfectly. Also the step from using this program to directing a real robot arm proved to be viable and the robot would perform a correct movement across the box.

When using the 3D camera to generate a point cloud, the surface presented in Rviz appeared to be close to perfect. However, small imperfections made it hard for Gödel to recognise the precise surface. The path planning across the recognised surface worked well, but since the recognised surface did not perfectly match the actual top of the box, executing the cut sometimes had a small offset resulting in cutting the cardboard instead of the tape. With the use of the rotating blade as an end effector this did not show to be a problem as the blade could cut through the cardboard effortlessly. Dents in the top of the box did cause a problem. Due to Gödel being a 2D path planner, the robot would always cut in a straight line. However if there would be any difference in height between different parts of the section that was to be cut, it could result in a part of the box not being cut. This problem could be explored more by using a 3D path planner.

Overall, with these results it can be said that it is possible to implement automation in the box cutting process. The main difficulties reside in the scanning. Mainly the accuracy of the 3D camera and the imperfections of the box will pose problems. Solutions for this can be explored by using a more accurate 3D camera and a 3D path planning package. Apart from this it can be concluded that this project has produced a satisfactory proof of concept.

References

[1] Nodes. (n.d.). In ROS Wiki. Retrieved Decemeber 19, 2019, from http://wiki.ros.org/Nodes

[2] Topics. (n.d.). In ROS Wiki. Retrieved December 19, 2019, from http://wiki.ros.org/Topics

[3] About ROS. (n.d.). In About ROS. Retrieved December 19, 2019, from http://www.ros.org/about-ros/

[4] Scan-N-Plan™. (n.d.). In Scan-N-Plan™. Retrieved December 19, 2019, from https://rosindustrial.org/scan-n-plan

[5] ROS-Industrial. (n.d.). In ROS-Industrial. Retrieved December 19, 2019, from https://rosindustrial.org/

VII. Appendices

Appendix 1:

Tutorial for altering the original Gödel application to a box cutting robot application

In this tutorial, explanation is given about how the original Gödel application is altered in order to create the box cutting robot application.

This tutorial is written for ROS (Robot Operating System) version Kinetic which is running on Ubuntu 16.0.4. This tutorial assumes that Ubuntu 16.0.4, ROS Kinetic and Scan-'n-Plan-library (specifically Gödel) are already installed and therefore lacks instructions for installing Ubuntu 16.0.4, ROS Kinetic and Gödel. This tutorial is written with the assumption that the reader has sufficient knowledge about C++ and ROS. Therefore, basic definitions are not explained. Next to that, basic operations like editing/adding variables in header files and how to source a workspace are not mentioned and assumed that the reader is able to perform these operations.

The project page of Gödel is: https://rosindustrial.org/scan-n-plan
The codes of the original Gödel project could be found on:
https://github.com/ros-industrial-consortium/godel

The codes of the box cutting robot application could be found at:
https://github.com/RLWBruins/boxcuttingrobot

Main objective 1: Simulation

Sub objective 2: Get a box point cloud into Godel

By default, there are three flat surfaces in Gödel. These surfaces are originally defined in the file *point_cloud_descriptions.yaml* in package *godel_irb1200_support*. Since the project is about cutting boxes, a box has to be loaded into Gödel. For the box cutting robot, two flat surfaces are removed and the other one is used to generate a box. The entire file was changed to the following:

```
frame_id: world_frame
point_cloud_descriptions:
 - x: -0.196
   y: 0.6175
   z: 0.162
   rx: 0.0
   ry: 0.0
   rz: 0.0
   l: 0.45
   w: 0.305
   h: 0.255
 resolution: 0.005
```

Sub objective 3: Insert the cutting tool as end effector

For the box cutting robot an ABB IRB1200 robot is used. The original end effector *blending_end_effector* was copied and renamed to *boxcutting_knife*. The file *blending_eff_macro.xacro* was renamed to *boxcutting_knife_macro.xacro*.

The file *Endeffectorcollision.STL* containing the end effector of this project was added in this package to the folders *meshes/visual* and *meshes/collision*.

Within the file *boxcutting_knife_macro.xacro*, the link *blending_tool* was changed so that the visual and collision tags contain *Endeffectorcollision.STL*. In all the other links (except the *blending_tool*-link because that is where geometry of the knife is located) all geometry-tags and its contents were removed. The links themselves can not be removed because Gödel assumes that they exist. If links are removed, there will be errors if links are removed and Gödel will not start.

The following parameters were changed to the following:

```
ensenso_optical_x: -0.055
ensenso_optical_y: -0.115
ensenso_optical_z: 0.068
```

keyence_tcp_x: 0.0
keyence_tcp_y: 0.025
keyence_tcp_z: 0.128

These coordinates indicate the point of where the 3D-camera (*ensenso_optical_x/y/z*) and the tip of the powered blade (*keyence_tcp_x/y/z*) are positioned relative to the link: *bracket*. Link: *bracket* is positioned at the end of the robotic arm.

The other parameters, which are set in this file are not relevant since only a modified version of the laser scanner path is used for the box cutting robot.

In order to change the blending tool to the box cutting knife tool, in the file *irb1200_workspace.xacro* (package: *godel_irb1200_support*) line 4 was changed to :

<xacro:include filename="$(find ***boxcutting_knife***)/urdf/***boxcutting_knife_macro***.xacro"/>

In Gödel, there is a demo cell around the robot arm which prevents the robot from moving outside this specified region. For the boxcutter it could be necessary to let the robot move outside this box. Therefore the demo cell should be enlarged. This is done by changing in the file a*utomate_demo_cell_macro.xacro.xacro* (package: *godel_irb1200_support*) the values of the parameters on lines 9, 10, 11, 13 and 14 to:

demo_cell_width: 10.0
demo_cell_length: 10.0
demo_cell_height: 10.0

robot_to_corner_x: -5.0
robot_to_corner_y: -5.0

Remember that the robotic arm does need sufficient space around it. If the arm is not able to move freely to its maximum reach, the demo cell should be sized accordingly.

Sub objective 4: Alter the laser scan path planning
For the application of the boxcutter, the laser scanning path had to be modified. The path should go through the middle of the surface instead of making zigzag-movements over the whole surface.

It appeared that this always happens when in the file *path_planning.yaml* (package: *path_planning_plugins*) the parameter *scan_width* parameter given has always exactly the same value as the width of the surface. However, this would mean that everytime this value has to be set manually for the right box width. In order to always make just one slice over the

surface, independent of the size of the surface, some changes were made in the file *profilometer_scan.cpp* (package: *path_planning_plugins*). This resulted in the robotic arm, crossing the centerline of the box once, across the length of the tp surface.

To make this happen, the following two changes have to be made in the file *profilometer_scan.cpp* (package: *path_planning_plugins*) :

- line 204 of the file has to be changed to :

  std::vector<RotatedRect> slices = sliceBoundingBox(bbox, ***bbox.h****, params.overlap);*

- In the function *sliceBoundingBox* the variable *n_slices* has to be set to 1. Thus, change the line number 75 containing

  n_slices = static_cast<int>(bbox.h / (adjusted_width)) + 1;

  to

  n_slices = 1;

These two changes let the application create just one path through the middle of a surface instead of creating a zigzag-path over the whole surface.

In the same file (*profilometer_scan.cpp*) the variable *GROWTH_FACTOR* is set equal to 1.2 (instead of the default value of 1.1). This makes the path 1.2 times longer than the length of the box. This is just a measure to make sure that the knife will entirely cross the whole surface and not does not stop before, for instance, reaching the end of the surface. There is a chance that this might happen because of surface recognition errors since a 3D-camera will be used which will not have perfect accuracy.

After the creation of the path through the middle of the upside of the box, the coordinates of the first point and the last point of this trajectory are determined. These points should lie on both outer edges of the box. The coordinates of these points are stored in the variables *beginCoordinatesX*, *beginCoordinatesY*, *endCoordinatesX* and *endCoordinatesY*. Also, a variable *distance* is created. The knife should travel a path with the length of 2.0**distance* (from y=-distance to y=+distance) along the outer edges.

All the points of the path through the middle of the upside of the box are stored in the variable: *points*. This is the same as in the original code of Gödel. The points of the path over the outer edges of the upside of the box are stored in the variabele *points2*. These points are stored separately because *Points2* contains all points for which the robotic arm has a yaw difference of -1.5707 radians (approximately equal to -90 degrees) compared to the points contained in the variable: *points*.

To make the procedure more clear, an overview is given of how the cutting tool moves:

- moves from scan position to coordinate (beginCoordinatesX;beginCoordinatesY;0.0)
- moves through the middle of the top surface to (endCoordinatesX;endCoordinatesY;0.0)
- is lifted 5 centimeters above top surface (endCoordinatesX;endCoordinatesY;0.05)
- moves to (beginCoordinatesX;beginCoordinatesY-distance;0.05) while rotation the cutting tool with -90 degrees over z-axis.
- the cutting tool is lowered with 5 centimeters to (beginCoordinatesX;beginCoordinatesY-distance;0.0)
- the cutting tool moves, while cutting, to (beginCoordinatesX;beginCoordinatesY+distance;0.0)
- is lifted 5 centimeters above top surface (beginCoordinatesX;beginCoordinatesY+distance;0.05)
- moves to (endCoordinatesX;endCoordinatesY-distance;0.05)
- the cutting tool is lowered with 5 centimeters to (endCoordinatesX;endCoordinatesY-distance;0.0)
- the cutting tool moves, while cutting, to (endCoordinatesX;endCoordinatesY+distance;0.0)
- is lifted 5 centimeters above top surface (endCoordinatesX;endCoordinatesY+distance;0.05)

To give a more technical overview, here is stated for variables *points* and *points2* which coordinates they contain. First, all points stored within variable *points* are executed. After that, all points stored within variable *points2* are executed.

Within variable *points* the following coordinates (listed in chronological order) are stored:

1. (beginCoordinatesX;beginCoordinatesY;0.0)
2. Points of the path between the coordinates (beginCoordinatesX;beginCoordinatesY;0.0) and (endCoordinatesX;endCoordinatesY;0.0). The points in this path are linearly positioned between these two coordinates, and have an increment of distance of *SCAN_APPROACH_STEP_DISTANCE* per point.
3. (endCoordinatesX;endCoordinatesY;0.0)
4. (endCoordinatesX;endCoordinatesY;0.05)

Within variable *points2* the following coordinates (listed in chronological order) are stored:

1. (beginCoordinatesX;beginCoordinatesY-distance;0.05) [While moving to this coordinate, the robotic arm also makes a -90 degrees rotation over z-axis.]
2. (beginCoordinatesX;beginCoordinatesY-distance;0.0)
3. Points of the path between (beginCoordinatesX;beginCoordinatesY-distance;0.0) and (beginCoordinatesX;beginCoordinatesY+distance;0.0). The points in this path are linearly positioned between these two coordinates, and have an increment of distance of size *SCAN_APPROACH_STEP_DISTANCE* per point.
4. (beginCoordinatesX;beginCoordinatesY+distance;0.0)

5. (beginCoordinatesX;beginCoordinatesY+distance;0.05)
6. (endCoordinatesX;endCoordinatesY-distance;0.05)
7. (endCoordinatesX;endCoordinatesY-distance;0.0)
8. Points of the path between (endCoordinatesX;endCoordinatesY-distance;0.0) and (endCoordinatesX;endCoordinatesY+distance;0.0). The points in this path are linearly positioned between these two points, and have an increment of distance of size *SCAN_APPROACH_STEP_DISTANCE* per point.
9. (endCoordinatesX;endCoordinatesY+distance;0.0)
10. (endCoordinatesX;endCoordinatesY+distance;0.05)

The complete code for function *generatePath* could be found on the Github page of the box cutting application (https://github.com/RLWBruins/boxcuttingrobot/blob/master/godel/path_planning_plugins/src/openveronoi/scan_planner.cpp).

Sub objective 5: Remove path planning process for blending

In the original Gödel code, the file *Blending_service_path_generation.cpp* (package: *godel_surface_detection*) contains a function called *generateProcessPath*. In this function the paths for the blending, the laser scan and the edge paths are generated. Since the box cutting robot application will only use the laser path, the generating process of the blending paths and the edge paths are commented. This results in the application generating only a laser scan path.

Sub objective 6: Select the top surface of the box

Only the top surface of a box is of interest for the box cutting robot application. In the original Gödel application, after the scan and surface detection process has been completed, the operator has to select manually which surface has to be cut. In order to automate the box cutting process, the selection of only the top surface has to be done automatically. The closer a surface is to the 3D camera, the more points the point cloud of that surface contains. Since the box cutting robot application makes a scan above the place were the box should be located, the point cloud of the top surface contains the most points. By selecting only the surface consisting of the highest number of points, the top surface of the box is selected.

This is done in the file *surface_detection.cpp* (package: *godel_surface_detection*) in the function *find_surfaces* between lines 373 and 418.

Sub objective 7: Execute the scanning

At this stage, the robot has a box cutting knife as end effector instead of the blending tool, Gödel will generate just one laser scanner path right through the middle of the surface and the knife moves, oriented perpendicular to the surface. In other words, at this point the basic functions of the box cutting robot are already working. However, the following problem occurs: By giving the robot the command of executing, it will show the following error message in the

terminal: *Keyence ROS server is not available on service /change_program*. This error occurs because Gödel assumes that there is a keyence laser scanner connected to the computer. The effect is that the execution of the robot will be cancelled (this is caused by the *return false;* lines). In this case there is not a real keyence laser scanner connected and it is also not needed for the box cutting robot. To overcome this problem, the following pieces of code are commented in the file *keyence_process_service.cpp* (package: *godel_process_execution*):

- ```
  if (!keyence_client_.exists()) {
      ROS_ERROR_STREAM("Keyence ROS server is not available on service " <<
      keyence_client_.getService());
      return false;
  }
  ```

- ```
  keyence_experimental::ChangeProgram keyence_srv;
  keyence_srv.request.program_no = KEYENCE_PROGRAM_LASER_ON;

  if (!keyence_client_.call(keyence_srv))  {
      ROS_ERROR_STREAM("Unable to activate keyence (program " <<
      KEYENCE_PROGRAM_LASER_ON << ").");

      return false;
  }
  ```

- ```
  keyence_srv.request.program_no = KEYENCE_PROGRAM_LASER_OFF;
  if (!keyence_client_.call(keyence_srv)) {
      ROS_ERROR_STREAM("Unable to de-activate keyence (program " <<
      KEYENCE_PROGRAM_LASER_OFF << ").");
      return false;
  }
  ```

Sub objective 8: Give the cutting tool the correct orientation

The original Gödel code lets the robotic arm move in such a way that the knife will not move perpendicular, but parallel to the surface. This is a very undesirable orientation of the robotic arm and knife for the box cutter. Therefore, the orientation has to be altered to make sure that the knife will be oriented perpendicular to the surface.

The following information has to be understood in order to understand and solve this problem. A path trajectory consists of a set of poses. Although the yellow trajectory displayed only shows the location, a pose consists of a location and an orientation. The orientation of the poses in the trajectory determines the orientation of the robotic arm and its end effector. By giving every pose the correct orientation, it is possible to let the end effector move, oriented perpendicular, across the surface.
  ```

Several changes have to be made in the file *scan_planner.cpp* (package: *path_planning_plugins*) in the function *generatePath* in order to alter the orientation of the pose. Changing the orientation of a pose could be done by using the following line:

```
pose.orientation =  tf::createQuaternionMsgFromRollPitchYaw(roll,pitch,yaw);
```

The orientation is saved as a quaternion message. With this line, Euler angles are converted into quaternion messages. Giving the roll a value of 1.5707 radians (is equal to 90 degrees) and the pitch a value of 0.0 radians (zero degrees) will let the knife move oriented downwards and thus perpendicular over the surface of a box.

The knife has to be rotated correctly over the z-axis. This is automatically done correctly by Gödel. Therefore, this value has to be kept the same. For that purpose, the value for the yaw-angle first has to be determined, before altering the orientation of the pose. This yaw-angle has to be used again to give the new orientation the same yaw-angle. Determining the Euler angles of the orientation of a pose is done by the following code:

```
tf::Quaternion q_original(pose.orientation.x, pose.orientation.y,
pose.orientation.z, pose.orientation.w);
tf::Matrix3x3 m(q_original);

double roll, pitch, yaw;
tf::Matrix3x3(q_original).getRPY(roll, pitch, yaw);
```

The original relevant piece of code in this file in function *generatePath* looks like:

```
std::transform(points.begin(), points.end(), std::back_inserter(scan_poses.poses),
               [boundary_pose_eigen] (const geometry_msgs::Point& point) {
  geometry_msgs::Pose pose;
  Eigen::Affine3d r = boundary_pose_eigen * Eigen::Translation3d(point.x, point.y,
point.z);
  tf::poseEigenToMsg(r, pose);
  return pose;
});
```

This piece of code has to be replaced by the following. It also contains some code which will be explained later in this appendix.

```
   std::transform(points.begin(), points.end(), std::back_inserter(scan_poses.poses),
                  [boundary_pose_eigen] (const geometry_msgs::Point& point) {
     geometry_msgs::Pose pose;
     Eigen::Affine3d r = boundary_pose_eigen * Eigen::Translation3d(point.x, point.y,
point.z);
     tf::poseEigenToMsg(r, pose);

     ros::NodeHandle nh;
     float angleCorrection = 0.0;
     nh.getParam("angleCorrection",angleCorrection);

     // Convert roll, pitch and yaw to Quaternion :
     float roll2 = 1.5707, pitch2 = 0, yaw2 = 0;

     ///* GET THE EULER ANGLES AND PUBLISH THEM
     tf::Quaternion q_original(pose.orientation.x, pose.orientation.y,
pose.orientation.z, pose.orientation.w);
     tf::Matrix3x3 m(q_original);

     double roll, pitch, yaw;
     tf::Matrix3x3(q_original).getRPY(roll, pitch, yaw);

     yaw2 = yaw+angleCorrection;


     pose.orientation =  tf::createQuaternionMsgFromRollPitchYaw(roll2,pitch2,yaw2);


     return pose;
   });
```

Main objective 2: Real robot arm

Sub objective 1: Establish connection with the ROS server on the robotic arm

Get the correct drivers and connect to the ROS server, which is running on the robot.

1. For this project an ABB IRB1200 robot was used at Robohouse (TU Delft). For the connection a library, which was written by a TU Delft employee was used. Therefore, Robohouse has its own tutorial on how to establish a connection with the robot. Because of that, it is not possible to write a general tutorial for this. However, to get it working at Robohouse, a set of packages called *ABB experimental* had to be installed. When building the workspace, an error will occur since there are two packages with the name *abb_irb1200_support*. Keep the just downloaded package and remove the package originally downloaded with Gödel.

2. When connection to the ROS server, installed on the ABB IRB1200 robot, is established (it is possible to check this by pinging to the robot and checking if any response is received) it is time for the next step. When running Gödel and trying to execute the robotic arm, an error will occur that the right kinematic plugin has not been found. To overcome this problem, open a new terminal window and execute the following command:

   sudo apt-get install ros-kinetic-trac-ik-kinematics-plugin

   This will install the correct kinematic plugin.

3. Now it will be possible to control the robot by executing the command:

   roslaunch abb_irb1200_5_90_moveit_config moveit_planner_execution.launch

   To make it work with Gödel, the following two changes were made in the file *moveit_planner_execution.launch* (package: *godel_irb1200_moveit_config*) :

   - Lines 34 up to and including line 52 was in the original Gödel code. But this has been commented.

   - At line 28 the following code has been added :

```
 <group unless="$(arg sim)">
   <include file="$(find
abb_irb1200_support)/launch/robot_interface_download_irb1200_5_90.launch"
>
     <arg name="robot_ip" value="$(arg robot_ip)"/>
```

```
  </include>
</group>
```

This piece of code comes from the file *moveit_planning_execution.launch* (package: *abb_irb1200_5_90_moveit_config*) which is included in the package *abb_experimental* which had to be downloaded according to the tutorial done in step 1.

Main objective 3: 3D camera

For this specific box cutting robot, an Asus Xtion camera is used. Therefore, this part of the tutorial is written for the Asus Xtion.

Sub objective 1: Select an appropriate 3D camera and install the correct drivers

As explained in the report, it is chosen to use the Asus Xtion camera. For the box cutting robot application, the library *openni2* is used as drivers to connect with the Asus Xtion. In order to install this library open a new terminal and execute the following command:

*sudo apt-get install ros-kinetic-openni**

This will automatically install the required drivers for the Asus XTion camera.

To test whether the camera is working correctly, execute the following command:

roslaunch openni2_launch openni2.launch

After executing this command, go to the drop down menu and select */camera/depth/image_rect* and inspect whether the streaming is correct.

It is also possible to visually check whether the output of the Asus Xtion-camera is correct by generating a point cloud. This is done by the following instructions:

1. Open a terminal and execute the following command:

   rosrun rviz rviz

2. Go to *Global Options* and change the *Fixed Frame* by clicking on the dropdown menu and select *camera_link*.

3. Click on *Add* and create a *PointCloud2*. Change the topic of this PointCloud2 to */camera/depth/points*.

Now it is possible to inspect the point cloud which is generated by the 3D-camera.

Sub objective 2: Set the correct scan pose

A scan pose is a pose (as explained earlier: this consists of a location and an orientation) from which a scan is made. In the original Gödel version, the scan positions are set in file *robot_scan.cpp* (package: *godel_surface_detection*) in the function *create_scan_trajectory*. For the box cutting project, this function has been copied and modified into another version. Therefore, a new function called *create_scan_trajectoryImproved* (instead of *create_scan_trajectory)* is created. In this new function, the scanpose used in the box cutting robot application is set. This function is called in the function *scan* in the same file. For our

application, just one scan from above is enough since it is only necessary to know the size of the upside of the box. The newly created function contains the following code and starts at line 337:

```
bool
RobotScan::create_scan_trajectoryImproved(std::vector<geometry_msgs::Pose>&
scan_poses,
                                          moveit_msgs::RobotTrajectory& scan_traj)
{
 // This function is created for the box-cutting project
 tf::Transform world_to_tcp    = tf::Transform::getIdentity();
 tf::Transform tcp_to_cam_tf   = tf::Transform::getIdentity();
  tf::Transform transformation1 = tf::Transform::getIdentity();
 tf::Transform transformation2 = tf::Transform::getIdentity();
 tf::Transform transformation3 = tf::Transform::getIdentity();
 tf::Transform transformation4 = tf::Transform::getIdentity();
  geometry_msgs::Pose pose;
  // Pose #1 : (just one scan pose is needed since one scan of the upside is
enough)
 transformation1.setRotation(tf::Quaternion(tf::Vector3(0, 1, 0),
3.14159265359/2.0)); // Rotation over y-axis within local frame
 transformation2.setRotation(tf::Quaternion(tf::Vector3(0, 0, 1),
3.14159265359)); // Rotation over z-axis within local frame
 transformation3.setOrigin(tf::Vector3(0.0, 0.55, 0.8)); // Translation within
world_frame
 transformation4.setRotation(tf::Quaternion(tf::Vector3(0, 0, 1),
-3.14159265359/2.0));// Rotation over z-axis within wold_frame
  world_to_tcp = transformation4 * transformation3 * transformation2 *
transformation1 * tcp_to_cam_tf.inverse();
 tf::poseTFToMsg(world_to_tcp, pose);
 scan_poses.push_back(pose);
  move_group_ptr_->setEndEffectorLink(params_.tcp_frame);

 return true;
}
```

Sub objective 3: Let the box cutting application make a 3D snapshot

1. Now the drivers are installed and the scan pose is set, the point cloud which is generated by the 3D-camera has to be implemented into Gödel. Several steps need to

be completed in order to implement it into Gödel. First of all, the drivers have to be launched simultaneously when Gödel is launched. The file *irb1200_blending.launch* has been copied and renamed to *irb1200_box_cutting.launch*. Launching the drivers along launching the box cutting robot application is done by the added line at line 64 in the file *irb1200_box_cutting.launch* (package: *godel_irb1200_support*) :

<include file="$(find openni2_launch)/launch/openni2.launch"/>

2. Now the drivers are launched when Gödel is launched, there will be a topic available called */camera/depth/points* which contains the point cloud data. The next task is to read and implement these point cloud data into Gödel. In Gödel, the node *generate_point_cloud_node* (in the file *point_cloud_generator_node.cpp* in package *godel_surface_detection*) is used to generate a fake point cloud of a box. This node first creates a perfect pointcloud of a 3D-box and then saves this point cloud in variable *full_cloud* (of type *PointCloud*). After this point cloud has been created, it publishes the point cloud in a while-loop to the topic *generated_cloud* which is remapped to the topic *sensor_point_cloud*. The remapping is defined in *IRB1200_box_cutting.launch* at line 65 which contains:

   <remap from="generated_cloud" to="sensor_point_cloud"/>

   The *sensor_point_cloud* topic is published in rviz and the data of this topic also used for the detection of surfaces.

3. When the scan pose has been reached, a signal should be created to let *point_cloud_generator_node* know that a 3D screenshot should be made. Because the screenshot will be made in another node (in the node *generate_point_cloud_node*) and there should be communication between these two nodes, the parameter server could be used for this purpose. This is done in the file *robot_scan.cpp* (package: *godel_surface_detection*) in the function *scan* by the following lines (line 278, 279 and 280) which were added for this purpose:

```
// Let the node point_cloud_generator_node know that it is time to make a 3D screenshot :
ros::NodeHandle nh("/");
nh.setParam("make3DScreenshot",true);
```

4. The topic that publishes a point cloud into RViz is called *sensor_point_cloud*. To implement the 3D-scanner, the point cloud data produced by the Asus Xtion have to be published to the topic *sensor_point_cloud*. An easy way this could be done is by modifying the earlier mentioned file named *point_cloud_generator_node.cpp* (package: *godel_surface_detection*). The file is modified in such a way that the point cloud,

generated by the 3D-camera, is constantly published in a while-loop to the topic: *generated_cloud* (which is remapped to *sensor_point_cloud*) instead of the perfect point cloud of the box.

To get this done, the following changes have been made to the file *point_cloud_generator_node.cpp* (package: *godel_surface_detection*):

- The already existing function *run:* has been changed to the following:

```
void run() {
   ros::NodeHandle nh;
   ros::Publisher cloud_publisher =
nh.advertise<sensor_msgs::PointCloud2>(POINT_CLOUD_TOPIC, 1);

   if (init()) {

     full_cloud_.clear();
     full_cloud_.header.frame_id = frame_id_;

     // ORIGINAL GODEL CODE : genenerate_cloud();

     ros::NodeHandle nh("/");
     nh.setParam("make3DScreenshot",false); // Define the parameter for
the first time

     bool make3DScreenshot=false;

     sensor_msgs::PointCloud2 msg;
     pcl::toROSMsg(full_cloud_, msg);

     ros::Duration loop_duration(0.4f);
     while (ros::ok()) {

       nh.getParam("make3DScreenshot",make3DScreenshot);

       if ( make3DScreenshot ) {

         make_screenshot(); // Make a 3D screenshot A.K.A. read data
3D-camera
```

```
          pcl::toROSMsg(full_cloud_, msg); // Create new message with the
new added pointcloud data

          nh.setParam("make3DScreenshot",false); // Reset parameter
        }

        msg.header.stamp = ros::Time::now() - loop_duration;
        cloud_publisher.publish(msg);
        loop_duration.sleep();

      }
    }
  }
```

In this piece of code, the value of ROS parameter *make3DScreenshot* is read and transferred to the boolean *make3DScreenshot*. If this variable is set to true, the node *generate_point_cloud_node* knows that it has to make a 3D screenshot at that moment and calls the function *make_screenshot*. Immediately after that is done, ROS parameter *make3DScreenshot* is set to false again.

- A new void function called *make_screenshot* has to be created. In this function, the topic */camera/depth/points* generated by the Asus Xtion is read just one time as if it is making a screenshot. The message containing the point cloud data are converted into a *PointCloud2* class. After that, the *PointCloud2* is converted to *PointCloud* class. However, the point cloud created by Asus Xtion will be positioned as if the camera is positioned at the origin of the *world_frame* and oriented straight ahead. This is not correct since the camera is positioned at the end effector and looking downwards from the scan pose-position. Therefore, the pointcloud has to be transformed correctly before publishing. This is done by applying a transformation matrix called *transformationMatrix*. This transformation matrix is formed by multiplying four matrices called *transformation1* (180 degrees rotation about local z-axis)*, transformation2* (180 degrees rotation about local y-axis)*, transformation3* (translation of the scan pose in the *world_frame* with 0.55 meters in y-direction and 0.8 meters in the z-direction) and *transformation4* (rotation of -90 degrees about z-axis with respect to the *world_frame*). If the transformation is done, the pointcloud is added to the variable *full_cloud*. The full code of the function *make_screenshot* is as follows:

```
void make_screenshot()
{
```

```
   full_cloud_.clear(); // Prevent that multiple 3D screenshots/clouds
merge with eachother

   sensor_msgs::PointCloud2ConstPtr msg =
ros::topic::waitForMessage<sensor_msgs::PointCloud2>(
       "/camera/depth/points", ros::Duration(20.0f));
   pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_ptr(new
pcl::PointCloud<pcl::PointXYZRGB>());


   // Box-cutting project : Convert message to PointCloud2
   pcl::PCLPointCloud2 pcl_pc2;
   pcl_conversions::toPCL(*msg,pcl_pc2);

   // Box-cutting project : Convert cloud_ptr (class PointCloud2) to
PointCloud
   pcl::PointCloud<pcl::PointXYZ>::Ptr temp_cloud(new
pcl::PointCloud<pcl::PointXYZ>);
   pcl::fromPCLPointCloud2(pcl_pc2,*temp_cloud);

   // Transform PointCloud to the frame of the 3D-camera :
   tf::Transform transformationMatrix = tf::Transform::getIdentity();

   // Make rotation over z-axis in the local frame
   tf::Transform transformation1 = tf::Transform::getIdentity();
   transformation1.setRotation(tf::Quaternion(tf::Vector3(0, 0, 1),
-3.14159265359));

   // Make rotation over y-axis in the local frame
   tf::Transform transformation2 = tf::Transform::getIdentity();
   transformation2.setRotation(tf::Quaternion(tf::Vector3(0, 1, 0),
3.14159265359));

   // Make an translation within the world_frame
   tf::Transform transformation3 = tf::Transform::getIdentity();
   transformation3.setOrigin(tf::Vector3(0.0, 0.55, 0.8));

   // Make rotation over z-axis in the world_frame
   tf::Transform transformation4 = tf::Transform::getIdentity();
```

```
   transformation4.setRotation(tf::Quaternion(tf::Vector3(0, 0, 1),
-3.14159265359/2.0));

   transformationMatrix = transformation4 * transformation3 *
transformation2 * transformation1;

   // Make the transforming actually happen
   Eigen::Affine3d eigen3d;
   tf::transformTFToEigen(transformationMatrix, eigen3d);
   pcl::transformPointCloud(*temp_cloud, *temp_cloud,
Eigen::Affine3f(eigen3d));

   full_cloud_ += *temp_cloud;

   full_cloud_.header.frame_id = frame_id_;
 }
```

Further implementation of the pointcloud is explained in the next section.

Sub objective 3: Reset parameters for surface detection

The first problem with running Gödel with the 3D-camera is that the surface detection does not work satisfactory anymore. This probably has something to do with the fact that Gödel has to work with imperfect point clouds generated by the 3D-camera instead of the perfect point clouds generated in *point_cloud_generator_node.cpp*. To fix this problem, the parameters for the surface detection have to be altered so that the surfaces are recognized again.

An important part of the surface detection happens in the file *surface_segmentation.cpp* (package: *godel_surface_detection*). Specifically in the function *computeSegments*. Here, segments of points are determined. These segments of points are in file *surface_detection.cpp* (package: *godel_surface_detection*) in function *find_surfaces* regarded as surfaces.

In the file *surface_segmentation.cpp* in function *computeSegments*, it could be seen that there are several parameters for the surface detection. A couple of these parameters are:

- *Minimum cluster size*
- *Maximum cluster size*
- *Number of neighbours*
- *Curvature threshold*

The first problem is that surfaces in general, and also the surfaces of the box, are not always detected. To make Gödel detect all surfaces, the *number of neighbours* and the *curvature threshold* are decreased. After this change, more surfaces and therefore also the surfaces of the real box are detected by Gödel.

Increasing the minimum cluster size filters out small and unwanted surfaces. Increasing the maximum cluster size enables Gödel to also detect the topside. It appeared that the topside of the box was not always detected as a surface. This has to do with the fact that the closer a surface is positioned to the 3D-camera, the more points this surface contains in the generated point cloud. With the default values for the parameter *Maximum cluster size*, the surface of the topside of the box to be scanned often contained too many points to be recognized as a surface. Increasing the value of maximum cluster size solves this problem.

For the box cutting project, the default values for these parameters are changed to:

- *Minimum cluster size*: 20000
- *Maximum cluster size*: 1000000
- *Number of neighbours*: 10
- *Curvature threshold*: 0.01

All values of these parameters are arbitrarily determined until the surface detection worked satisfactory. With the new values for these parameters, the surface detection works correctly in the box cutting application. The result is that only the topside of the box is detected and all other surfaces are neglected.

Sub objective 4: Solve path planning deviation due to imperfect point cloud

The second problem which arose with working with imperfect point clouds is that the laser scan path has a small rotation regarding to the surface. It appeared that the point cloud of the surface is a bit rotated within its own frame (see figure 1). To fix this problem, the rotation of the point cloud within its own frame is calculated in the function *calculateAngleCorrection*. After this angle has been calculated, the point cloud is rotated back within its own frame with the same angle so that there is no rotation anymore. All these operations happen (and the necessary functions are called) in the function *generatePath* in the file *scan_planner.cpp* (package: *path_planning_plugins*).

The function *calculateAngleCorrection* works as follows. First, the point within the point cloud with the biggest distance towards the midpoint of this point cloud (averageX,averageY) and with coordinates x<0 and y>0 is determined. The distance is calculated by using the Pythagorean theorem and this is done in the function *calculateDistance*. The coordinates of this point are stored in mostLeftUpX and mostLeftUpY. According to the same principle, this happens to determine the point with coordinates x<0 and y<0 and the biggest distance towards the midpoint (averageX,averageY). The coordinates of this point are stored in mostLeftDownX and mostLeftDownY. Then, the angle of the triangle consisting of the points (mostLeftDownX;mostLeftDownY), (mostLeftUpX;mostLeftUpY) and

(mostLeftDownX;mostLeftUpY) is calculated. This angle is an approximation of the angle with which the point cloud is rotated within its own frame. After this angle has been calculated, all points within the point cloud are rotated back with a value of this angle. This solution gives satisfactory results for the box cutting project (see figure 2).

The function *calculateAngleCorrection* consists of the following code:

```
float openveronoi::ScanPlanner::calculateAngleCorrection
(std::unique_ptr<mesh_importer::MeshImporter>& mesh_importer_ptr, const
pcl::PolygonMesh& input_mesh) {

 // Zet PointCloud2 om naar PointCloud
 pcl::PointCloud<pcl::PointXYZ>::Ptr points(new pcl::PointCloud<pcl::PointXYZ>);
 pcl::fromPCLPointCloud2(input_mesh.cloud, *points);
 pcl::transformPointCloud(*points, *points,
mesh_importer_ptr->plane_frame_.inverse());

 double mostLeftDownX = 0.0;
 double mostLeftDownY = 0.0;
 double mostLeftUpX = 0.0;
 double mostLeftUpY = 0.0;
 double mostRightDownX = 0.0;
 double mostRightDownY = 0.0;
 double mostRightUpX = 0.0;
 double mostRightUpY = 0.0;

 double averageX = 0.0;
 double averageY = 0.0;

 for (int i = 0; i < points->points.size(); ++i) {
   averageX += points->points[i].x;
   averageY += points->points[i].y;
 }

 averageX /= (double)points->points.size();
 averageY /= (double)points->points.size();


 for (int i = 0; i < points->points.size(); ++i) {
```

```
   double x = points->points[i].x - averageX;
   double y = points->points[i].y - averageY;

   double distance = calculateDistance(x,0,y,0);

   if ( x < 0 && y < 0 && distance >
calculateDistance(mostLeftDownX,0,mostLeftDownY,0) ) {
     mostLeftDownX = x;
     mostLeftDownY = y;
   }
   if ( x > 0 && y < 0 && distance >
calculateDistance(mostRightDownX,0,mostRightDownY,0) ) {
     mostRightDownX = x;
     mostRightDownY = y;
   }
   if ( x < 0 && y > 0 && distance > calculateDistance(mostLeftUpX,0,mostLeftUpY,0) )
{
     mostLeftUpX = x;
     mostLeftUpY = y;
   }
   if ( x > 0 && y > 0 && distance > calculateDistance(mostRightUpX,0,mostRightUpY,0)
) {
     mostRightUpX = x;
     mostRightUpY = y;
   }
 }

 float angle = atan2(mostLeftDownY - mostLeftUpY, mostLeftDownX - mostLeftUpX) * 180.0
/ 3.14159265359 + 90.0;

 return 3.14159265359/180.0*angle;
}
```

Giving the points of the laser scan path the extra angle with a value of *angleCorrect* has been done by lines 267 until 286 in the same file:

```
   for ( int i = 0; i < points.size(); i++ ) {

     double radius  = sqrt(pow(points[i].x, 2) + pow(points[i].y, 2));
```

```
    float  angle    = atan2(points[i].y, points[i].x)+angleCorrection;

    points[i].x = radius * cos(angle);
    points[i].y = radius * sin(angle);

  }

  for ( int i = 0; i < points2.size(); i++ ) {

    double radius = sqrt(pow(points2[i].x, 2) + pow(points2[i].y, 2));
    float  angle  = atan2(points2[i].y, points2[i].x)+angleCorrection;

    points2[i].x = radius * cos(angle);
    points2[i].y = radius * sin(angle);

  }
```

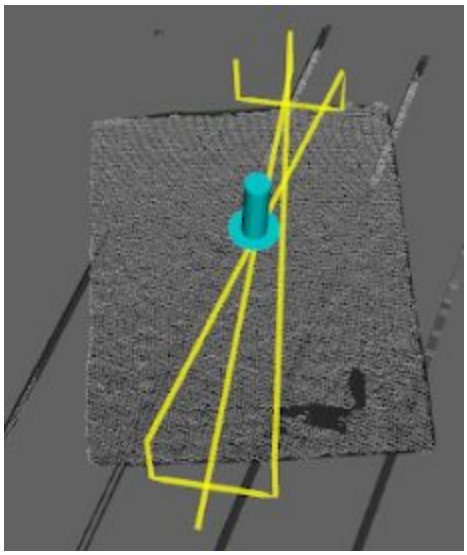

Figure 1 *Path is wrongly rotated*

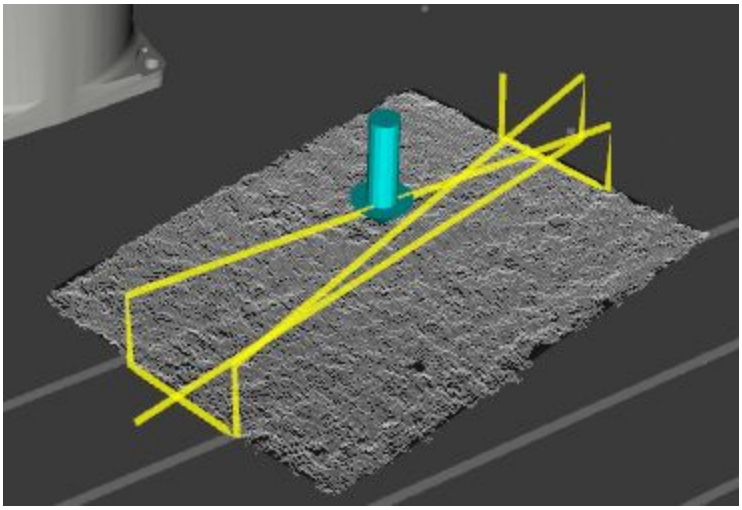

Figure 2 *The wrong path rotation has been corrected.*

After solving this last problem the ABB IRB1200 robot should work as a box cutting robot.

The box cutting robot should be working correctly by running the following commands :

- Run simulation:

roslaunch godel_irb1200_support irb1200_box_cutting.launch

- Run while connected to real robotic arm:

    roslaunch godel_irb1200_support irb1200_box_cutting.launch sim_robot:=false robot_ip:=[robot ip]

Appendix 2:

Implementing the GP25 robotic arm:

In this tutorial, it is assumed that the box cutting robot application is installed in the folder *~/catkin_ws*.

Step 1: download the entire package https://github.com/fizyr/yaskawa_gp25_support. Place this package in a place in the workspace that makes sense, e.g. *~/catkin_ws/src/abb*

Step 2: Change all names of the joints and links in the .xacro file of the GP25 (this file is called gp25_macro.xacro) to the names used in the irb2400 robotic arm. In the downloaded GP25 robotic arm all links and joint have names as *link_s* and *link_t* etc. The irb2400 has names as *link_0* and *link_1*. Therefore, changed these names of the links in the GP25 to the names of the links in the irb2400. So in the end you need to end up with link names such as *link_1 link_2* instead of *link_s* and *link_t*.

Step 3: Go to the map *config,* open the file *joint_names_gp25.yaml* and change the content with:

controller_joint_names: ['joint_1', 'joint_2', 'joint_3', 'joint_4', 'joint_5', 'joint_6']

Step 4: make a copy of the map *godel_irb2400_support* (location : *~/catkin_ws/src/godel/godel_robots/abb/godel_irb2400/godel_irb2400_support*) and name this *godel_yaskawa_gp25_support*. In the map *godel_yaskawa_gp25_support/urdf* change the name of the file *irb2400_workspace.xacro* to *gp25_workspace.xacro*.

Step 5: change in *CMakeList.txt* and *package.xml* the name *godel_irb2400_support* to *godel_yaskawa_gp25_support*.

Step 6: Go the the map *urdf* and open the file *gp25_workspace.xacro* and change line 3 to:

<xacro:include filename="$(find **yaskawa_gp25_support**)/urdf/**gp25_macro**.xacro"/>

Step 7: go the the map *launch*, copy the file *irb2400_blending.launch* and call it *gp25_blending.launch*. You can delete the file *irb2400_blending.launch*.

Step 8: open the file *gp25_blending.launch*, scroll down and change the line:

<include file="$(find godel_irb2400_moveit_config)/launch/moveit_planning_execution.launch">

By

*<include file="$(find **godel_yaskawa_gp25_moveit_config**)/launch/moveit_planning_execution.launch">*

Step 9: If you build the entire workspace, the robotic arm should be in it and it can also move. There is however still a couple of things that should be changed to get the movements behave like you want them to. The way the robotic arm is implemented right now will still cause the arm to make odd movements with blending and scanning. To repair this, the next changed have to be made. Open the file *gp25_macro.xacro* and remove in this file at joints 3, 4, 5 and 6 like down here the minus sign:

<axis xyz="0 -1 0"/> or *<axis xyz="0 0 -1"/>*

Should be changed to:

<axis xyz="0 1 0"/> or *<axis xyz="0 0 1"/>*

This makes sure that all the axis are mirrored. With the irb2400 this doesn't happen. By removing the minus sign it will now behave the same way as with the irb2400. If you run Gödel right now, the movements of the arm will probably behave better than before, but still as expected.

To be sure you can also change in between the <*limit*> tags the parameter "effort" from 100 to 0. In the IRB2400 it has the same values.

Still the movements aren't 100% accurate. This is because the GP25's origin and the IRB2400's origin probably have an offset respectively with each other. You can fix this to change at *joint_1* in the file *gp25_macro.xacro* the line

<origin xyz="0 0 0" />

To

<origin xyz="-0.05 0 0.11" />

This makes sure the offset is removed and the arm works fine right now with the exception that the base of the arm and the arm itself are now not coupled anymore. This is probably not the proper method to fix this problem but it is a workaround.

Step 10: build the workspace with the new package in it again by using the following commands:

cd ~/catkin_ws
catkin build
source devel/setup.bash [Change this or it will not detec tthe package]

The MoveIt!-configuration still isn't correct because it still uses the MoveIt!-configuration from the workspace with the irb2400. The right MoveIt!-configuration will be made in the following steps.

Create MoveIt!-configuration:

Making the URDF-file:

To make the MoveIt!-configuration for the new workspace with the GP25 you need a .urdf-file from the entire workspace. So far there's just .xacro-files. These have to be converted into .urdf-files.

To fix this, follow the next steps:

1. Put this in your terminal: *cd ~/catkin_ws/src/godel/godel_robots/abb/godel_irb2400/godel_yaskawa_gp25_support/urdf/*
2. Put this in your terminal: *rosrun xacro xacro --inorder -o gp25_workspace.urdf gp25_workspace.xacro*
3. The *.urdf*-file can be loaded in the MoveIt! Setup Assistant
4. To be sure if the *.urdf*-file is still valid you can use the following command:
   Put this in your terminal: *check_urdf gp25_workspace.urdf*

MoveIt! Setup Assistant :

1. In the directory *~/catkin_ws/src/godel/godel_robots/abb/godel_irb2400/* Make a new directory with the name *godel_yaskawa_gp25_moveit_config*. Leave this direcetory empty, because the setup assistant is going to fill it with config files.

2. Start the setup by putting the following line in the terminal: *roslaunch moveit_setup_assistant setup_assistant.launch*

3. Load the earlier made *gp25_workspace.urdf* from the directory *~/catkin_ws/src/godel/godel_robots/abb/godel_irb2400/godel_yaskawa_gp25_support/urdf/* in the MoveIt! Setup Assistant and make sure everything is set the same way as for the *godel_irb2400_moveit_config*. These settings can be found in the config files of the *godel_irb2400_moveit_config*. You can copy them from these files, but to make things easier it is summed up down here. Make sure you copy the values to the same ones as listed down here:

   - Tab *Self Collisions* :
     - Click just on “Generate Collision Matrix”
   - Tab *Virtual joints* :

    - Virtual Joint Name : FixedBase
    - Child Link : world_frame
    - Parent Frame : map
- Tab *Planning Groups :*
    - Group #1 :
        - Group name : manipulator
        - Base link : base_link
        - Tip link : tool0
        - Kinematic solver : kdl_kinematics_plugin/KDLKinematicsPlugin
    - Group #2 :
        - Group name : manipulator_asus
        - Base link : base_link
        - Tip link : kinect2_move_frame
        - Kinematic solver : kdl_kinematics_plugin/KDLKinematicsPlugin
    - Group #3 :
        - Group name : manipulator_ensenso
        - Base link : base_link
        - Tip link : ensenso_sensor_optical_frame
        - Kinematic solver : kdl_kinematics_plugin/KDLKinematicsPlugin
    - Group #4 :
        - Group name : manipulator_keyence
        - Base link : base_link
        - Tip link : keyence_tcp_frame
        - Kinematic solver : kdl_kinematics_plugin/KDLKinematicsPlugin
    - Group #5 :
        - Group name : manipulator_tcp
        - Base link : base_link
        - Tip link : tcp_frame
        - Kinematic solver : kdl_kinematics_plugin/KDLKinematicsPlugin
- Tab *Robot Poses* :
    - Do nothing and skip.
- Tab *End Effectors* :
    - Do nothing and skip.
- Tab *Passive Joints* :
    - Do nothing and skip.
- Tab *ROS Control* :
    - Do nothing and skip.
- Tab *Simulation* :
    - Do nothing and skip.
- Tab *3D Perception* :
    - Chose : Point Cloud. After that do nothing and skip.
- Tab *Author Information* :

- Fill in legit or random information, otherwise the MoveIt! Setup cannot be finished.

4. If everything is set properly, select in the last tab in the directory *godel_yaswaka_gp25_moveit_config* and make sure it places all files in this directory.

5. If you have followed these steps, there are still a few files missing that were used with the *godel_irb2400*. Put the file *moveit_planning_execution.launch* in the directory *godel_yaskawa_gp25_moveit_config/launch*. This *.launch* file can be found in the directory *godel_irb2400_moveit_config/launch*. Change in this file all the references to the *irb2400* to those of the *gp25.*

6. Change in the file *moveit_planning_execution.launch* the follow line (if that isn't already done in *step 5*):

   <rosparam command="load" file="$(find abb_irb2400_support)/config/joint_names_irb2400.yaml"/>

   by :

   <rosparam command="load" file="$(find yaskawa_gp25_support)/config/joint_names_gp25.yaml"/>

7. Make the following adjustments. Change in the directory *godel_yaswaka_gp25_moveit_config/launch* the files *move_group.launch* and *irb2400_workspace_moveit_controller_manager.launch.xlm* by the files with the same name from the directory *godel_irb2400_moveit_config/launch*. Change if needed all references from the *godel_irb2400_…* directory to the *godel_yaskawa_…* directory. Keep all the names in the workspace equal to those in the *irb2400_workspace.* Copy the file *controllers.yaml* from *godel_irb2400_moveit_config/config* to *godel_yaswaka_gp25_moveit_config/config*.

8. Build the workspace with the new package in it again with the following commands:

   - In the terminal: *cd ~/catkin_ws*
   - In the terminal: *catkin build*
   - In the terminal: *source devel/setup.bash* [otherwise it will not find the new package]

9. If these steps are followed it should work if you put the following command in the terminal:

   roslaunch godel_yaskawa_gp25_support gp25_blending.launch