NTNU
Norwegian University of
Science and Technology

3D Inspection for quality control with Robot Manipulator

Using open source software and freeware to
make an automatic 3D scanning application

Magnus Molaug Seines

Master of Science in Cybernetics and Robotics
Submission date: June 2017
Supervisor: Jan Tommy Gravdahl, ITK
Co-supervisor: Esten Grøtli, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Summary

As modern industry is automated the need for robots performing complex tasks increases. To achieve fully automated production both making the product and inspecting it for quality assurance needs to be automated. Robotic manufacturing processes have been around for a long time and has greatly increased the efficiency of modern industry. In this thesis existing framework for automatic 3D inspection is examined, and an attempt at making a working 3D inspection implementation is made. The work is centered around offline inspection planning, using a model reference to generate a path for fully inspecting a 3D object with a sensor attached to a robot arm. Continuing the work in Seines (2016), the inspection planning algorithm proposed by Bircher et al. (2015a) is adapted further in an attempt to make it optimal for planning inspection of small to medium sized objects with an industrial robot arm.

The modification of the structural inspection planning algorithm is evaluated compared against the algorithm without modification by comparing resulting path length and algorithm convergence. The augmentations appear to increase the performance of the inspection planning algorithm when planning in 5 dimensions.

To simulate inspection with a UR5 robot, a mount for the Intel RealSense SR300 sensor is 3D printed and attached to the wrist link. Sensor position relative to the wrist joint is measured and used to make a custom URDF and SRDF for the UR5 such that collision between the sensor and other joints can be prevented and sensor position can be controlled accurately.

The resulting inspection paths are simulated in Gazebo with two different approaches to motion planning. To further test in detail how effective individual improvements to the algorithm have been with regards to inspection path quality the algorithm is separated into different iterative versions, where the path generated by each version with otherwise identical parameters are simulated, and the resulting motion of the robot is compared with the intended path.

The augmentations of the inspection planner improve robot behavior. However, logging sensor state reveals the sensor has incorrect orientation when reaching the waypoints. The Cartesian motion planner fails to produce a full trajectory for most paths. The inverse kinematics based point to point planner leads to undesirable behavior such as arcing motions between waypoints and goal overshoot. To test basic functionality for point cloud acquisition some of the generated paths are run on the real robot, while both the robot state and 3D sensor output is being logged.

The SR300 performed poorly, possibly due to poor lighting conditions.

It is concluded that the inspection planner should be rewritten to incorporate trajectory planning in joint space. The inspection planner appears to perform well but needs additional measures for adapting the path for inspection with manipulators.

Sammendrag

Moderne industri blir mer og mer automatisert, og behovet for roboter som kan utføre komplekse oppgaver øker. For å kunne ha helt automatisk produksjon må alle deler av produksjonen automatiseres. I denne oppgaven er fokus på å implementere automatisk kvalitetskontroll av produktet ved hjelp av en robot arm og et dybdekamera. Oppgaven tar først for seg eksisterende implementasjoner av ulike 3D inspeksjonsalgoritmer, og evaluerer hvorvidt fremgangsmåten kan kopieres for dette problemet.

Arbeidet er sentrert rundt videre utvikling av inspeksjonsplanleggings algoritmen foreslått av Bircher et al. (2015a) for å tilpasse den til å planlegge inspeksjonsbane for små objekter ved hjelp av en UR5 robot med en Intel RealSense SR300 sensor festet i enden.

Effekten av utvidelsene av algoritmen blir evaluert i forhold til algoritmen uten utvidelser. Konvergens og hvorvidt algoritmen genererer optimale inspeksjonsbaner undersøkes og de genererte inspeksjonsbanene blir sammenlignet basert på lengde. Utvidelsene av algoritmen ser ut til å ha positiv effekt for planlegging av inspeksjonsbaner med 5 frihetsgrader.

Det 3D printes et feste for 3D sensoren, og sensorens posisjon i forhold til enden av roboten måles og brukes til å konfigurere URDF og SRDF filer slik at roboten kan unngå kollision mellom sensoren og omgivelsene.

Inspeksjonsbanene blir så simmulert i Gazebo for å teste hvordan roboten oppfører seg under inspeksjon og for å sammenligne to ulike fremgangsmåter for å kontrollere roboten. Utvidelsene av inspeksjonsbane planleggings algoritmen deles opp i iterative versoner, og forbedring i robotens oppførsel fra verson til verson testes.

Enkelte av inspeksjonsbanene testes på UR5 roboten for å teste hvorvidt inspeksjonsbanene lar 3D sensoren inspisere objektene nøye nok til å gjenskape dem fra målingene.

3D sensoren ga ikke gode nok målinger, men det kan være verdt å teste igjen under bedre lysforhold.

Den kartesiske bevegelses planleggeren slet med å gjennomføre mesteparten av inspeksjonsbanene. Punkt til punkt bevegelses planleggeren hadde til tider uforutsigbare og voldsome bevegelser mellom punkter langs banen, og førte konsistent til stier der sensoren bevegde seg forbi målet. Under både simmuleringer og eksperimenter klarte ikke noen av bevegelses planleggerene å få sensoren til å nå ønsket orientering. Feil i hvordan sensor posisjon og orientering måles kan være skylden til feilen, selv om målingene for posisjon virker korrekte.

Den utvidede inspeksjonsbane planleggeren lager gode baner, men at den manglende koblingen mellom inspeksjonsbane og robot bane er mindre optimal. Fen foreslåtte veien videre er å holde på fremgangsmåten for å plassere sensoren, men å skrive om infrastukturen i inspeksjonsbane planleggeren slik at den gir baner som kan bevege roboten dirrekte.

Preface

Thanks to my supervisors for being patient and helping me when i got stuck, and my friends for enduring any complaints i had after bad days at the office.

Oh man, I am so sick of weird ROS errors...

Table of Contents

List of Tables

List of Figures

Abbreviations

TCP	=	Tool Center Point
RRT	=	Rapidly Exploring Random Tree
TSP	=	Traveling salesman problem
AGP	=	Art Gallery Problem.
URDF	=	Universal Robot Description Format
SRDF	=	Semantics Robot Description Format
ROS	=	Robot Operating System
PRM	=	Probabilistic Road Map
OMPL	=	Open Motion Planning Library

Chapter 1

Introduction

As modern industry is automated the need for robots performing complex tasks increases. To achieve fully automated production both making the product and inspecting it for quality assurance needs to be automated. Robotic manufacturing processes have been around for a long time and has greatly increased the efficiency of modern industry. In this thesis existing framework for automatic 3D inspection is examined, and an attempt at making a working 3D inspection implementation is made. The work is centered around offline inspection planning, using a model reference to generate a path for fully inspecting a 3D object with a sensor attached to a robot arm.

This thesis will be continuing the work from Seines (2016) of adapting an inspection planning algorithm by Bircher et al. (2015a) for inspection with a robotic manipulator. The algorithm is originally intended to be used with aerial drones for inspection of buildings.

1.1 Problem description

The intended use case is an assembly line quality inspection. To compare a manufactured object against a digital reference model, sensor measurements must be accurate enough to reconstruct the object accurately. This implies the inspection path must provide sufficient amount of measurements to generate a dense point cloud of accurate depth measurements. Assuming the inspection takes place on an assembly line, minimizing the amount of time for inspection while still maintaining high accuracy inspection is desired.

To achieve this the goal the approach chosen in this paper is to compute an ordered lists of viewpoints, where each viewpoint represents a position of the 3D sensor, in x,y,z coordinates, and Euler angles roll, pitch and yaw. This inspection path is automatically generated given a 3D mesh of the object being inspected in ASCII STL format, its location relative to the robot base, and sensor specs such as operating distance and field of vision. Automatic path generation requires solving a multi-dimensional constraint satisfaction problem for placing viewpoints able to observe the entire object. As well as finding the optimal path for the manipulator traversing the viewpoints. Repeated inspection means minimizing total joint movement is of interest as well.

In this paper, we are both looking to optimize the path generation as well as point to point movement by researching possible motion planning approaches. Both inspection planning and simulations are implemented for use with the Robot Operating System (ROS) framework, and as such motion planners will mainly use functionality from MoveIt and the Open Motion Planning Library(OMPL).

1.2 Contributions

This paper focuses on the further improvement of the path planning algorithm proposed by Bircher et al. (2015a). Improved usability for manipulators instead of aerial vehicles has already been started in Seines (2016). Contributions in this paper include adding detection of self-collision and exploring measures to achieve a more optimal path given the movement capabilities of a robotic manipulator, with the use of a fast inverse kinematics solver.

Previous work Seines (2016) succeeded in including a variable pitch, however, simulating the resulting paths proved less successful. Not only did the paths often include configurations bordering self-collision, several viewpoints where outside the workspace of the robot, or in positions with no feasible solutions to the inverse kinematics. Therefore a significant focus in this paper is to integrate manipulator kinematics into the path planning in an attempt to generate paths more suited for the manipulator.

More specifically this includes checking the placement of viewpoints and expanding with a randomized configuration sampler heavily inspired by the redundant roadmap approach to motion planning.

Setting up a full simulation in Gazebo, as well as implementing multiple approaches of motion control of the manipulator, following an ordered set of waypoints. Executing a select number of the paths on a real robot to compare the trajectory to the trajectory when simulating the robot.

Functions for logging 3D sensor point cloud frames, robot joint positions and sensor position real time. As well as logging and transformation of point clouds to align in a fixed frame.

1.3 Framework and practical considerations

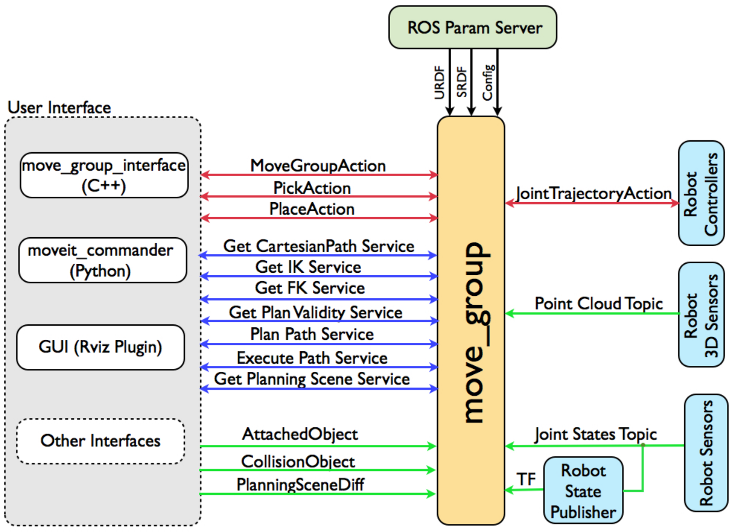
This paper builds heavily on the work done in previous project assignment, as the work done in Seines (2016) heavily incentives to keep using ROS.

1.3.1 ROS and MoveIt functionality explained

Robot Operating System(ROS) is an open source collaboration with the intention of providing a large, general toolkit for robot programmers. ROS also provides a simulation and motion planning framework that is applicable for multiple different robots regardless of function and manufacturer in the form of Gazebo and MoveIt. It is meant to act as a collaborative robot development platform. Thus many of the repositories used are often still in development and are patched regularly. This means functions are prone to change and tutorials that are up to date are sparse.

ROS works as a virtual machine keeping track of a parameter server, available topics, services, and actions. With this it is possible to coordinate multiple processes, possibly running on different computers or processors, to communicate and request services of one another.

Figure 1.1 Moveit system architecture concept. Picture from moveit home page: `http://moveit.ros.org/documentation/concepts/`



The functionality provided by MoveIt is based on the move group node, a multi-purpose node that provides a multitude of services designed to simplify robot control and interaction. Concept depicted in 1.1. The Move Group node provides a control layer between the user and the sensor and/or robot drivers.

The Move Group node perform its functions by keeping internal models of both environments and of the robot. The internal models contain information on the behavior of the robot, collision matrices, transformation data, joint type, as well as user defined functionality of the robot.

This is typically stored in multiple variables with predetermined names on the ROS parameter server. Meaning the Move Group node needs this information already uploaded or as a part of a robot specific move_ group launch script.

As the Move Group node keeps track of the environment it is capable of performing both point-to-point and multigoal trajectory planning, while avoiding known obstacles. During execution of a path, it has functionality for checking whether the goal is reached or not and will cancel a trajectory if it detects a collision along the path.

Why ROS and MoveIt?

The work done in this paper will continue using the ROS framework, with MoveIt as a control interface for both Gazebo and the real robot. As writing a lot of the code necessary for my projects again would be time-consuming and mastering the use of ROS has several benefits, as ROS is made to be a general purpose robotics control tool.

There are multiple arguments for and against using ROS in a student project.

Pro	Con
Existing code for complex problems	Steep learning curve
Seamless transition between simulation and real robot.	Packages with experimental status
Clean and easy to use interface for logging sensor data.	General lack of tutorials.

Table 1.1: Pros and cons of using ROS and MoveIt!

As previous work with ROS (Seines (2016)) mitigates the problem of the learning curve and a lack of tutorials, the positives are considered to outweigh the negatives.

Alternative approach

ROS is an open source project, therefore most of its functionality is publicly available and can be installed without installing ROS. Popular planning algorithms such as RRT* are publicly available, and implementing planning frameworks specifically for the application allows for more control of robot behavior. Robot drivers such as the UR Modern Driver made by Andersen (2015) are publicly available and can be run without ROS installed. The modern driver provides a UR Script control interface. Allowing for control by streaming scripts to the robot instead of generating general purpose trajectory messages. Very little functionality has to be implemented to use this control interface as the URscript language contains functions for linear movement in both joint space and tool space.

For simulation, there are a dozen different robot simulation tools out there., such as VisualComponents, a simulation program that specifies in being able to simulate full assembly line functionality. It is also possible to use general-purpose simulation tools such as LabView, or Matlab.

1.3.2 Robot setup

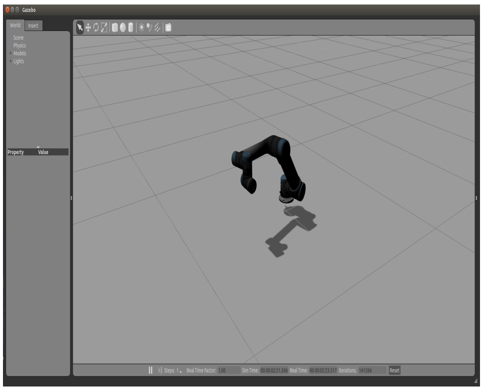
The robot of choice is the UR5 from Universal Robots. It is a small robot arm designed to be safe and easy to use, and while it is less accurate than some industrial robots, less power means less risk of crushing either the object it inspects or the 3D sensor. The weight of the camera is minimal, meaning the arm does not need to have a lot of lifting power.

The ur5 struggles with frequent self-collision, however, it should be possible to work around this issue with careful motion planning.

Simulations

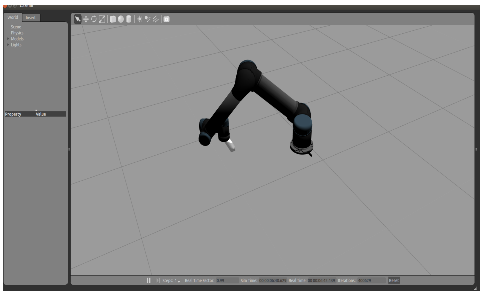
Installing ROS Indigo and the universal robots package, you get access to simulate the robot in Gazebo. Gazebo is a general purpose robot simulation program, able to load any robot given its description file. (URDF)

Figure 1.2 Gazebo simulation,depicted ur5 without end effector



Adding tools to the robot can be done by modifying the robot description file (URDF). The sensor is added by adding two extra fixed joints and two links to the official robot description found in the ur_ description package.

Figure 1.3 Gazebo simulation,depicted ur5 with and effector



Experiments

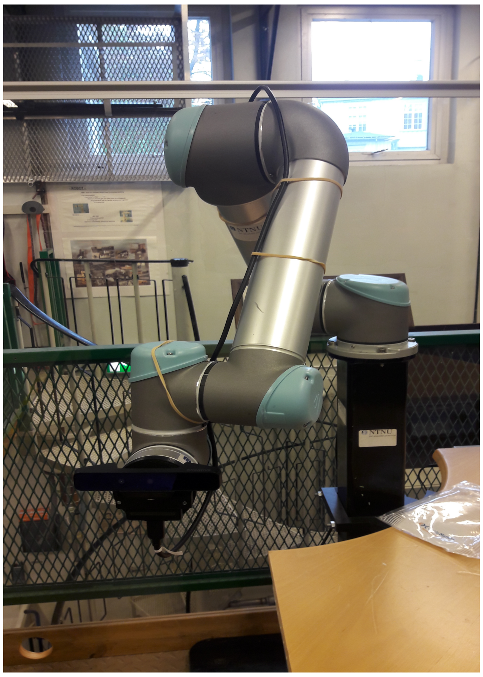
Path viability is examined by comparing the paths resulting from the unmodified algorithm with the paths resulting from the algorithm after the possible improvements discussed in this paper.

To test if the paths are feasible for inspection with the robot, they are first simulated in Gazebo. The position and orientation of the sensor are logged and compared with the viewpoints from the path to determine the quality of the motion planning approached.

A few paths are executed on the robot to attempt inspecting objects placed on the table in front of the robot. Sensor position and orientation are logged to investigate trajectory quality.

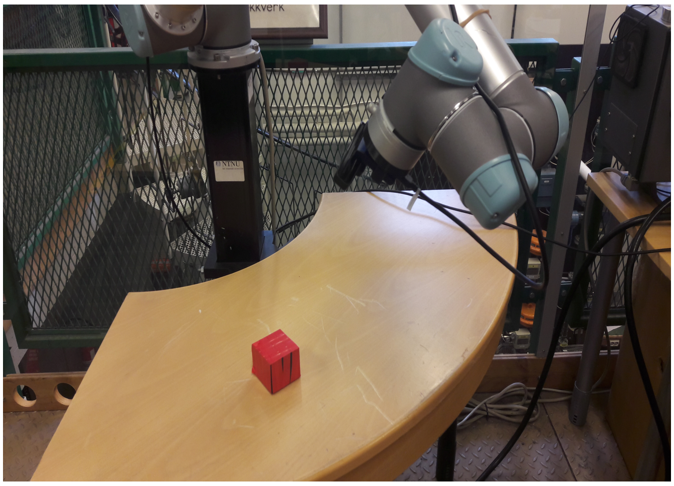
Sensor output from robot inspection is examined, and possibilities for reliable quality control is discussed.

Figure 1.4 Robot setup.



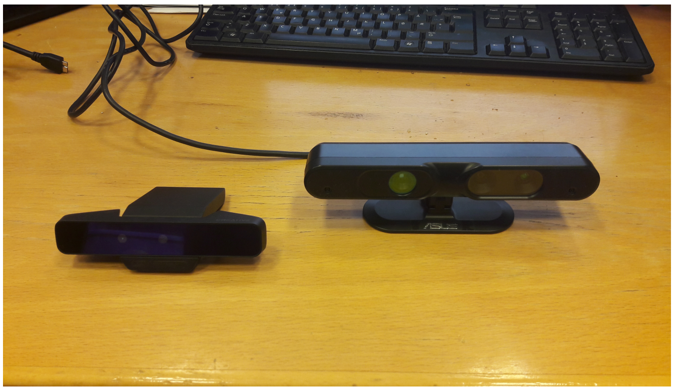
The objects are placed as far from the wall as possible to maximize the robots available workspace.

Figure 1.5 Inspection of cube, covered in red tape in an attempt to increase sensor visibility



1.3.3 Choice of 3D measurement sensor

Figure 1.6 Intel RealSense SR300 (left) and ASUS Xtion Pro(right)



There were several concerns when selecting a 3D sensor. Firstly our inspection planning algorithm assumes data as a continuous stream with a sensor capturing depth in a cone. Thus sensors using time of flight, projected grids or stereo camera solutions are the most viable.

Another concern is budget and time since the focus of the experiments are to validate inspection optimally and the amount of surface covered by a path, not pushing maximal millimeter accuracy. The priority will thus not be to spend several thousand dollars on expensive top end industrial scanners, with no online price listing.

ASUS xtion pro

A cheap budget alternative is the ASUS Xtion Pro, While it is widely used for different applications, it operates on a recommended distance of 0.8 to 2.5 meters. This is a problem because the UR5 has a length of 85 cm from base to the tooltip, meaning the tool center point will have difficulties being more than 40 cm away from the object consistently.

While the details in the back are captured, the object is prone to not registering. Also, notice the deformation and choppiness of the object edges. For industrial use, this is simply not accurate enough. In addition, the quality looks increasingly worse the closer the sensor moves towards the object.

Intel sense

Another budget option is the Intel RealSense SR300. It operates on a recommended distance of 0.2 to 1.2 meters. Thus the optimal operation region is fitting when accounting for the length of the UR5.

At first, glance using the demo program supplied by Intel (used to take these screen caps) looks very promising compared to the ASUS. The much higher point density makes it possible to construct high-quality models from the scan. The scanner conserves the shape of the cup with what appears to be smoother edges.

Figure 1.7 Screenshot of ASUS depth data from a coffee cup at 50 cm distance, yellow implies depth data and black is undetermined depth

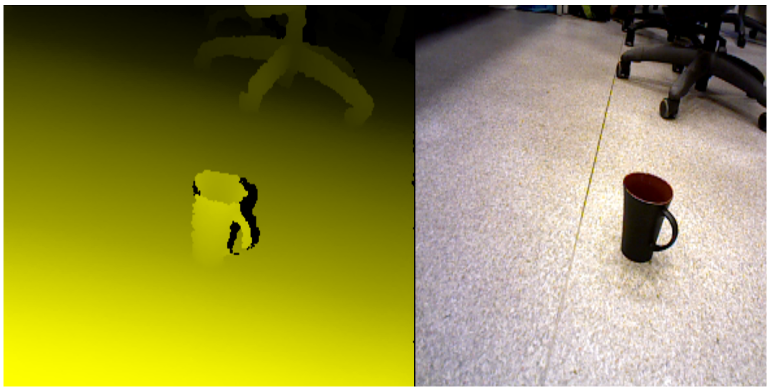
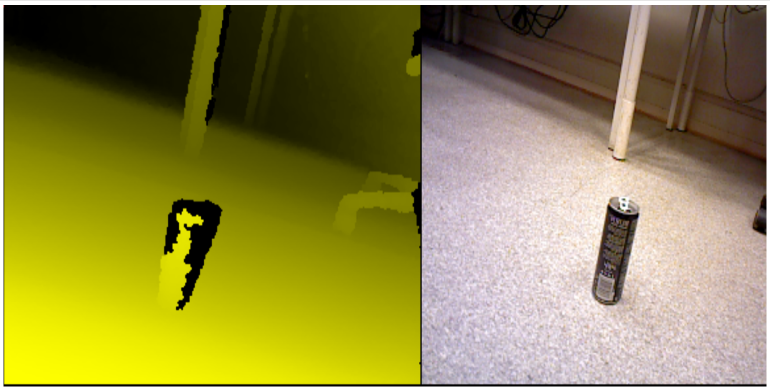


Figure 1.8 Screenshot of ASUS depth data from a box at approximately 50 cm distance, yellow implies depth data and black is undetermined depth.



However, the sensor appears to have some issues, like higher sensitivity to light and a higher percentage of points not capturing valid depth data at medium to long range.

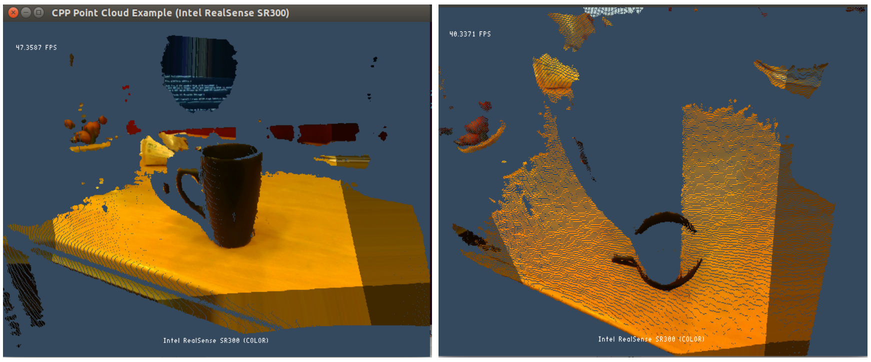
It still favored as in the range of 15 to 40 cm it captures a high amount of data where the ASUS Xtion Pro struggles when it gets closer to the object. In addition to this the RealSense camera is considerably smaller, as well as it has a hole to fasten it with a standard camera screw.

Ideal sensor

For an ideal sensor for industrial quality control existing products being marked for industrial inspection create a baseline.

- Artec Spyder
- Geoscan

Figure 1.9 Screenshot of Realsense depth data from a coffee cup at 40 cm distance, picture to the left rotated to a view from the top


CPP Point Cloud Example (Intel RealSense SR300)
47.3587 FPS
Intel RealSense SR300 (COLOR)
40.3371 FPS
Intel RealSense SR300 (COLOR)

- MetraScan 750-R

`https://www.artec3d.com/3d-scanner/artec-spider`
`http://geoscan.nl/over-ons/3dscanservice/`
`https://www.creaform3d.com/en/metrology-solutions/optical-3d-scanner-met`

These different sensors all promise more than 0.1 mm accuracy. At most all the way down to 0.064 mm. The sensors come with software to reconstruct a model from the measurements, thus work out of the box. The price is the only limiting factor. These sensors have a price of 30 000 $ to 50 000 $.

Most heavy duty industrial scanners come with additional tech beyond the scanner. Examples include positioning aids, custom lighting or additional cameras. The sensors generally estimate distance based on laser measurements or deformation of grids made out of high-frequency structured light.

Looking at marketing demos there is a long way to go before the setup presented in this paper will be competitive on the professional marked. Youtube video of industrial robot scanning a complex object in real-time for approximately a minute with the MetraSCAN 750-R, full scan timed at approximately 2 minutes. `https://www.youtube.com/watch?v=QGvoF9hT2kM`

Chapter 2

Literature Review

Automatic 3D scanning is a complex problem with many factors to consider, in this report the focus is on path planning for 3D inspection, or more specifically attempting to solve automated path generation and inspection of 3D objects with known shape. There is a lot of articles covering the subject of 3D scanning, with the focus on one of the themes further explained below, often incorporating a full system or simulation of a full system.

Coverage path planning, view planning, inspection planning are all expressions used to describe the process of planning the position and orientation of a sensor to fully cover a provided set of objects that need inspection. While the sensor has to yield to obstacles, the point to point movement is often simplified to be straight lines and circular motions.

While a motion planning algorithm typically handles the movement following the inspection plan point to point while yielding to constraints on numerical accuracy of the pose, maximum translational velocity, and angular velocity.

To achieve an optimal solution the coverage path planner should take account of the limitations of the robot intended to move the sensor around as well as how much space around obstacles is necessary to avoid risk when using expensive sensors.

2.1 Existing implementations

Section copied from Seines (2016).

2.1.1 Inspection with manipulator

The idea to use robotic manipulators for quality assurance by scanning the manufactured object is not a new idea. The proposed applications of 3D scanning include generating CAD models for objects without a known model, for preservation and restoration purposes or product inspection for assembly line applications. Most implementations use a sensor attached to the end of an industrial manipulator and estimate depth by triangulation with laser (Kriegel et al. (2015)) or structured light projectors where deformation of a fixed grid or pattern is used to estimate distance(Vasquez-Gomez et al. (2014), Wu et al. (2014), Krainin et al. (2011)). Karaszewski et al. (2016) tests how a multitude of NBV algorithms perform on inspection of museum objects. And testing the result using various accuracy structured light based scanners.

Most papers found with a manipulator and sensor setup utilize a next best view inspection planning algorithm to achieve full coverage. As a common assumption is that a reference model is unavailable. The focus of this literature is often on detailed CAD model generation.

The method proposed by Kriegel et al. (2015) using a laser striper attached to a Kuka KR16 industrial robot **??** achieves an accuracy varying from $0.64mm$ to $1.5mm$ depending on the shape being scanned. Where the accuracy measured is the least squared error when comparing the model with a hand scanned model using a very accurate scanning tool. The method and experimental setup of Wu et al. (2014), using an Artec Spider with an accuracy of up to $0.1mm$, however using their approach the object generally has to be scanned over several iterations to fill in details. However, their formula for calculating resulting model accuracy, and its metric is not specified.

The company behind the MetraScan sensors has a youtube video showing a demo where inspection of an object is done real time with their scanner and an industrial robot. It is unclear whether or not the path is generated automatically or programmed by hand by engineers. However, it apparently inspects a complex object accurately in approximately 2 minutes. `https://www.youtube.com/watch?v=QGvoF9hT2kM`

2.1.2 Inspection with unmanned vehicles or drones

There are also several papers on 3D inspection planning for unmanned vehicles or drones. Proposed applications are often in structure inspection from air or underwater and more detailed inspection of parts of ships or outdoor structures.

The algorithm featured in the work this paper expands upon (Seines (2016)) is proposed by Bircher et al. (2015a) and is implemented to use a rotorcraft or fixed-wing UAV to inspect large structures or areas in their examples. With a custom stereo camera sensor as the depth measurement tool of choice. Englot and Hover (2017) proposes a Redundant Road Map (RRM) planner and demonstrates its uses by inspecting ships hull with an autonomous underwater vehicle.

2.2 Coverage path planning

In this paper, coverage path planning is used to describe the process of calculating sensor positions to sufficiently cover an object.

2.2.1 Offline coverage path planning

For industrial use, it is likely that a manufactured product is made with a 3D model reference. Therefore a model dependent algorithm should be ideal compared to an exploratory algorithm, as a model dependent algorithm will have computational overhead while scanning.

If real-time 3D model reconstruction achieved, holes in the model can be filled by revisiting viewpoints known to observe missing areas of the reconstructed mesh.

The structural inspection planner proposed by Bircher et al. (2015a) is an offline planner for structural inspection with either a rotorcraft or a fixed wing drone. Their method utilizes a stereo camera setup and assumes that the depth sensor captures data in a pyramid shaped cone. The algorithm first solves a constraint satisfaction problem placing a viewpoint observing each part, or triangle, of the input mesh. After this, the TSP is solved to find the shortest path connecting these points. This is repeated for a set number of iterations. With each iteration, the viewpoints are updated to minimize the distance between the preceding and succeeding viewpoints of the previous path. In this way, the inspection path is iteratively updated. The source code for the planner is open source and utilizes RRT* for point to point planning in the proximity of obstacles and an LKH based solver for the Traveling Salesman sub-Problem

The redundant roadmap planner proposed by Englot and Hover (2017) (also Englot and Hover (2011)) is an offline planner, exemplified by using the algorithm to plan the inspection of a ship hull by an unmanned underwater vehicle. The algorithm starts by constructing a roadmap by adding configurations until each point is observed a certain number of times, hence the term redundant. The configurations are sampled in the neighborhood of the object and kept if it is collision free and has a free line of sight towards one or more discrete parts of the model. The configuration that adds unique observations is selected, and this continues until each part of the model has at least been observed by one configuration. By adding post processing or heuristics, this process can be augmented to have higher priory to add configurations close to already added configurations, or the path between existing configurations.

Galceran and Carreras (2013) defines coverage path planning and explains multiple popular approaches to the problem. The explanations are quite superficial but show how both simple and complex problems can be approached. The article refers to papers using random sampling based methods to cover complex 3D structures (Englot and Hover (2011)), genetic algorithm (Jimenez et al. (2007)) for full coverage as well as literature on coverage with the use of multiple robots.

2.2.2 Model independent scanning algorithms

The more common approach in literature, when planning with industrial robot arms, is model independent scanning. Robot independent scanning is algorithms exploring the object, filling it out to make a continuous surface while constructing a mesh from the point cloud in real-time. The papers usually focus on the application of reconstruction and object preservation for archaeologists, or model generation for use in 3D modeling or film etc. The object shape and exact size are not considered known in advance.

Next Best View/Scan (NBV/NBS) algorithm variants are widely used for view planning of unknown objects. Next Best View(NBV) is a greedy algorithm concept, where the algorithm is using feedback from real-time model generation and sensor feedback to estimate where the sensor should be placed next. The algorithms are usually driven by heuristics deciding if it is optimal to keep exploring the object, or if it is more desirable to fill details of the existing models given the number of scans already performed. This is to ensure the algorithm produces a detailed model after a finite number of scans.

Kriegel et al. (2015) use a method based on boundary detection and shape estimation to estimate what parts of the model are missing. On-line boundary detection is implemented by iterating over newly acquired parts of the mesh identifying edges in the generated triangle mesh without any bordering triangles. Surface trends are used to estimate object curvature aiding scanner placement. The 3D sensor of choice is a 3D laser stripe measuring depth by triangulation. A scan is represented as a start and end position with a fixed sensor orientation. Different candidates are selected by a utility function depended on how much area they are expected to cover and distance from the current position, as well as execution time. The utility function has objective values for exploration and modeling, dependent the number of scans performed so far. For modeling small distances are preferred to fill out holes in the scanned surface. While for exploration, maximizing covered area is preferred to discover the object shape as fast as possible.

Krainin et al. (2011) inspects a model by gripping the object and turning it in front of the camera, using an NBV inspired algorithm for planning gripper orientations.

2.2.3 Path optimization and post processing

The art gallery problem is the problem exemplified by covering an art gallery by a minimal number of observers. In this case a solution of the art gallery problem mean inspecting the object with a minimal number of viewpoints. Concerning coverage of the object, an inspection path defined by a minimal number of sensor positions is likely not to be desired. However, when considering efficiency with respect to time, a path represented by a minimal number of waypoints is ideal. The Triangle Inequality states that the total length between point A and B is always shorter or equal to the distance from point A to C plus the distance from C to B. Thus removing redundant viewpoints should reduce path length. If the robot has to slow down or stop to ensure the sensor position and orientation is correct at each waypoint, fewer waypoints mean less overhead.

Path smoothing algorithms benefit inspection, as continuous motion will lead to a smooth motion of the sensor. For industrial application, minimizing jerk and actuator stress is vital for minimizing manipulator stress and wear.

Englot and Hover (2012) proposes an extension of the RRT* algorithm by adding a heuristic smoothing and simplifying the resulting path. The algorithm is tested by employing an AUV to inspect the running gear of two ships. The algorithm starts by removing redundant waypoints, as in waypoints not uniquely observing a part of the object. Then the heuristic will prioritize waypoints forming paths along smooth lines. As the heuristic is implemented in the RRT* algorithm, it reduces the time needed to find the connecting path.

The algorithm proposed by Bircher et al. (2015a) has minimal post processing, as it relies on forming a smooth path when iteratively improving the viewpoints. With additional measures ensuring linear inspection paths when planning inspection with fixed wing AUVs. While the algorithm is said to minimize the number of viewpoints in the plan by solving the Art Gallery Problem, previous experience with the planner Seines (2016) shows that output paths will contain the same amount of waypoints as the number of triangles forming the input mesh. Implementing a way to remove redundant waypoints likely to improve path quality.

Gasparetto and Zanotto (2007) proposes an algorithm for smoothing trajectory velocity, acceleration, and jerk. The algorithm focuses on generating a trajectory with continuous acceleration for each joint. Starting with a path defined in operating space, kinematic inversion is applied to get a path in joint space. The optimal path from point to point is found iteratively by an SQP solver by minimization of an objective function penalizing squared jerk over the trajectory. Yang and Sukkarieh (2010) proposes an algorithm for fitting a path to an ordered number of waypoints in 3D space to satisfy curvature constraints. By projecting the trajectory onto two 2D spaces, applying changes and then transforming back to 3D space. With the goal of generating paths with minimal joint jerk.

2.3 Motion planning

As the solution to the coverage planning problem, in this case, is a set of goal positions providing the required sensor positions for full coverage of the object being inspected, the motion between each point is not necessarily specified. In an attempt to optimize the time used for inspection, the accuracy the path is followed and maximize the time where the sensor is making use full observations. A motion planner resulting in the end effector and manipulator motion providing maximum stability of the 3D sensor and minimize the time spent inspection is desirable.

There are several approaches to motion planning available. ROS, via MoveIt, provides a large variety of general purpose motion planners from the Open Motion Planning Library(OMPL). This includes implementation of Rapidly-exploring Random Tree algorithms such as RRT* and RRTConnect. Probabilistic RoadMap algorithms, and projection based motion planners. It also provides an interface for collision detection, kinematic solvers, and communication with robot drives through the Move Group node.

To plan robot trajectories from poses in tool space it is required to have a kinematic solver to translate between operation space and joint space. A collision checker capable of checking whether a given joint configuration has self-collision or collision with the environment, and a motion planner that can calculate trajectories between two states.

Saha et al. (2006) proposes a planning algorithm suited for inspection and welding with an industrial robot. The algorithm assumes that multiple poses in tool space have to be reached, but the order in which they are reached is not of importance. It is assumed that it is desired to reach the points in minimal time and robot motion. The algorithm uses a heuristic based path generation algorithm for generating close to optimal paths from goal to goal, and a bi-directional PRM planner for generating the full trajectory.

Descartes is an experimental path planning package for ROS (found at `https://github.com/ros-industrial-consortium/descartes`) with the purpose of providing a high-level interface for robot manipulation. The package is made specifically for applications where the robot needs to move the end effector linearly between multiple poses defined in tool space. The package includes a dense motion planning algorithm that returns a robot trajectory optimized for speed, minimal joint movement or energy consumption given a list of waypoints as input. The algorithm requires a specific inverse kinematics interface, and solves the shortest and optimal path by computing all possible inverse kinematics solutions for each goal point, and optimizing linear movement between the goals.

2.3.1 inverse kinematics

Since the inspection planning algorithm returns plans in tool space and not joint space, a vital part of planning the robot trajectory is an inverse kinematics solver to translate way-points into joint space. There are multiple approaches to solving the inverse kinematics, with the focus on maximizing either accuracy, speed or providing additional functionality such as returning the configuration closest to a given seed.

A widely used kinematics application worldwide is the Kinematics and Dynamics Libraries (KDL `http://www.orocos.org/kdl`) by the Orocos(Open Robot Control Software) Project. Using kinematic chains to represent robots and providing real-time forward and inverse kinematics solvers.

While widely used as a generic kinematics solver, KDL has some issues. Beeson and Ames (2015) attempts to solve some of this issues in ”trac_ ik”. The solvers are set up similarly as to those in KDL, where the main difference is the inclusion of seeded states and fixes for false negatives in some of KDL’s functionality.

The inverse kinematics solver used in the ROS package *ur_kinematics* are made by Hawkins (2013) from Georgia Tech. The work was motivated by the conventional solutions being both slow and returning wrong solutions at times. This application is an extremely efficient analytic solver.

2.3.2 Point to point planning algorithms

There are a multitude of algorithms for point to point planning in multi-dimensional configuration space. Most are implemented as general purpose algorithms able to plan trajectories in both three and six dimensions, needing only a function for calculating distance and checking for collisions.

RRTConnect, proposed by Kuffner and LaValle (2000) is a single query path planning algorithm focused on returning a viable path quickly. The algorithm initializes two Rapidly-expanding Random Trees with the two states. The trees are expanded by adding collision free states with linear, collision-free paths to an existing state until the two trees intersect. When a connection between the trees are found the resulting path is returned. As such this algorithm Can not be expected to return an optimal path. RRTConnect is the default planner selected when planning for a UR5 in ROS using the Move Group interface.

Karaman and Frazzoli (2011) performs an analysis of popular probabilistic planners such as RRT and PRM and proposes three asymptotically optimal path planning algorithms in the form of RRT*, PRM* and RRG. These algorithms are extensions of the discussed probabilistic planners.

PRM or Probabilistic Road Map is a multiple query planning algorithm consisting of a pre-processing phase and a query phase. In the processing phase, the roadmap is put together by adding a given number of random states and collision free connections with already existing nodes as edges to the road map. Further path planning queries are computed based on the assembled road map. The PRM algorithm is effective in scenarios where the environment is static and the robot has to perform repetitive motions. RRT* is an expansion of the RRT algorithm, where asymptotic optimally is reached by adding edges not only to the node a new state is expanded from, but also to any state within a given radius. To keep the tree structure, whenever a new state is added, it is added in a way that the branch it is connected to forms the shortest path from the root to the new node, of the available paths. After adding a new node redundant edges, as in edges not part of an optimal path, are removed.

Akbaripour and Masehian (2016) proposes a semi lazy probabilistic roadmap algorithm. Under the roadmap construction, random configurations are added to the roadmap, until it is sufficiently rich. However, the end effector is the only link being checked for collisions. Then each node is connected to the k closest nodes, with a distance corresponding to a weighted distance of the difference. The shortest path is computed by adding both start state and goal state to the map and using Dijkstra to compute the shortest path.

Due to the fact that the widely used random planners are either asymptotically optimal and require a lot of time computing optimal solutions, others are only probabilistic complete, meaning they will return a solution if it exists and fail if not, however, there are no guarantees that the solution is optimal with any given margin. Thus a common approach is to add augmentations where the path is optimized through the use of heuristics or post processing algorithms.

Nonprobabilistic approaches include general purpose path search algorithms, adapted for use with the given robot. Erdős et al. (2016) uses an A* search to generate a collision-free path.

2.3.3 3D model reconstruction

Both the Intel Realsense SR300 and the Asus Xtion Pro outputs sensor measurements as point clouds. While the point cloud data can be used directly for quality inspection (see subsection), it may be necessary to reconstruct the object from the point cloud to compare some qualities, such as object curvature. A detailed and accurate model, reconstructed from inspection measurements, can be used to prove inspection accuracy, and possibly validate coverage percentage, and the sensors ability to detect details.

ROS has a designated library for point cloud utility functions, PCL Rusu and Cousins (2011), that adds several features for processing point cloud data. The library contains functions for processing, filtering and transforming different point cloud data types it can be used as a framework for point cloud acquisition and model reconstruction.

Marton et al. (2009) proposes a greedy triangulation algorithm for connecting the points in a point cloud together into triangles. The triangles are combined to form a 3D model. Surface normals are estimated and maintained during reconstruction to retain object shape. Example code for its intended use can be found at `http://pointclouds.org/documentation/tutorials/greedy_projection.php`.

Iterative Closest Point(ICP) is an algorithm that can be used for fitting point clouds together. The ICP algorithm attempts to match a point cloud to an another set of points by minimizing the total distance between the sets. Working frame by frame, the first frame serves as a basis, then iteratively each subsequent frame is added with a matching transform given by the ICP algorithm, the resulting point cloud is filtered between each iteration. The points can be connected with connecting each point to its closest k neighbors and filling each completed shape formed this way.

There are multiple possible modifications to the ICP algorithm Cappelletto et al. (2013) adds color to the square error to improve the surface generation when using a 3D camera with color and depth output.

Using sensor output for quality control

The subsection is copied from Seines (2016).

The resulting point cloud data from a scan is typically transformed to match a fixed frame and then used for reconstructing a 3D model of the object. The resulting model is then compared to a reference to measure quality as a function of how well the models match.

These methods often apply orientation matching strategies, such as PCA. Principal component analysis fits the measured data onto an orthogonal set of vectors, where the first vector defines the axis with the highest measurement variance and has a length proportional to this variance. Additional vectors are added as the vectors defining the axes with highest measurement variance orthogonal to the vectors already in the set. By computing the transform required to match the orthogonal base for each model, it is likely that the same transform will make the two objects have matching orientation and scale. Principal Component Analysis(PCA) is explained in detail Jolliffe (2002)

Tangelder and Veltkamp (2003) proposes a method of comparing similarities between two polyhedra. First Principal Component Analysis(PCA) if performed to match object orientation. Secondly, the objects are divided into a grid. Each object is assigned a signature, based on the properties of each area of the grid. Possible properties include Gaussian curvature and variation of surface normals. These signatures are then compared by calculating Proportional Transportation distance. The Proportional Transportation distance is the estimate of the amount of work required to transform a set of weighted points to resemble another set.

Another method for comparing polyhedra is proposed by Tuzikov et al. (2000). It is rather complex and covers a similarity measure between two convex polyhedra. Based on Minkowski addition, as well as Volume relations between the two polyhedra.

Reconstructing a 3D model is not necessary for inspection as shown byBergström (2016) (As well as Bergström and Edlund (2016)). Bergström (2016) demonstrates an application an ICP based fitting algorithm in real-time. By fitting a point cloud onto a reference model after performing a Principal Component Analysis(PCA) to match orientations, it is possible to perform an effective on-line error measurement by using the fitting error of the point cloud. This is used to distinguish what object is being observed in the demo by assuming it is the one with the least error `https://www.youtube.com/watch?v=lm7_mwpOk0E&feature=youtu.be`, as well as to detect deformation of the object surface, exemplified by a small piece of paper in the video `https://www.youtube.com/watch?v=cPS-DY9sCz4&feature=youtu.be`.

Chapter 3

Theory

This paper is building on previous project work Seines (2016), where the structural inspection planner by Bircher et al. (2015b) was expanded to include variable pitch. Sections 3.1 to 3.1.5 are taken from Seines (2016).

This section will go into detail on the structural inspection algorithm and describe expansions to the algorithm intended to adapt it for planning inspection of small objects by a manipulator with a 3D sensor attached as an end effector.

The focus will be on possible modifications and extensions to the algorithm resulting in more manipulator friendly paths. This chapter will have a detailed look on an alternative model based approach, as well as have a detailed look on motion planning for a manipulator, to have the output path more closely resemble the resulting path when simulating it with a manipulator.

3.1 Structural inspection planner, finding viewpoints

Algorithm overview. In case two viewpoints cannot be connected linearly because of

Algorithm 1 Inspection path planner

```
 1: $k \leftarrow 0$
 2: Sample initial viewpoint configurations
 3: Compute cost matrix for the TSP solver
 4: Solve TSP to obtain initial tour
 5: **while** running **do**
 6:     Resample viewpoint configurations
 7:     Recompute cost matrix
 8:     Recompute best tour
 9:     Update best tour cost if applicable
10:     $k \leftarrow k + 1$
    **return** $T_{best}, c_{best}$
```

obstacles or similar issues, the algorithm uses the RTT* method by Karaman and Frazzoli (2011) to connect the points and estimate a cost. To compute the optimal tour the algorithm uses a Lin-Kernighan-Helsgaun Heuristic based TSP solver by Helsgaun (2000) Camera position is formulated as a QP and solved by the QP solver implemented in qpOASES (see Ferreau et al. (2014)), which features an efficient implementation of the active set algorithm.

3.1.1 Path computation and cost estimation

An initial tour is made by connecting the viewpoints found by solving the Constraint Satisfaction Problem(CSP) for each triangle in the input mesh. The tour is improved each iteration, according to an objective function determined by the order and value of the viewpoints from the preceding tour. Motion between viewpoints is assumed to be linear in absence of collision objects. To guarantee linear motion between each viewpoint does not exit the workspace, the workspace has to be locally convex. The linear motion from point g_{k-1} to g_k can be described as:

$$g(t) = (1 - s)g_{k-1} + sg_k$$

For s from 0 to 1.

Collision avoidance

The algorithm employs lazy collision detection for simplicity, using the RRT* algorithm for point to point planning if the workspace is obstacle heavy or for viewpoints in the presence of obstacles.

Obstacles are represented as boxes with variable dimension, position and orientation.

When updating viewpoints the algorithm does not actively avoid obstacles. The position minimizing the QP is checked for collisions. If a collision is found, the position is used to initialize an RRT structure. The tree is expanded until a collision free state satisfying the constraints of the QP is reached. If no feasible state is found the planner will exit assuming the obstacle is blocking feasible inspection of the triangle.

3.1.2 Viewpoint sampling

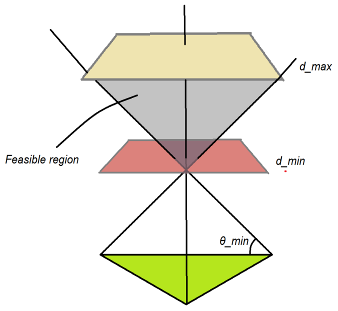
For each iteration of the planning algorithm, the optimal sensor positions are sampled first, based on the inspection tour from the previous iteration. Followed by optimization of orientation. Constraints on position must, therefore, be selected in a way that makes all feasible positions able to observe the triangle.

Position constraints independent of craft orientation

Assume the input mesh is made up of multiple triangles. Let the camera position be given by $g = [x, y, z]$. Corners of the triangle given by the vector $x_i, i = 1, 2, 3$, and the unit normal of the triangle is given by a_N.

Distance constraints (d_{min}, d_{max}) and the minimal incidence angle (θ_{min}), is selected to guarantee that the point selected is within the 3D sensors ideal range and will observe the triangle with an angle allowing for accurate measurements of the entire triangular surface.

Figure 3.1 Feasible region defined by min incidence angle and min and max distance



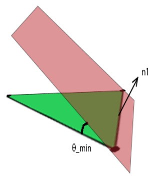
The minimal incidence angle constraint is given by 3 planes defined by each of the triangle edges and the triangle normal rotated (θ_{min}) around the same edge. Let $R_v(\theta)$ be a matrix representing a rotation of θ around the vector(axis) v and $n_i, i = 1, 2, 3$ be the normal of these planes. Then the line given by $g - x_i$ is at least at an angle of θ_{min} from the triangle surface if.

$$(g - x_i)^T n_i \geq 0, i = 1, 2, 3$$

$$n_i = R_{e_i}(\theta_{min}) a_N, i = 1, 2, 3$$

$$e_1 = (x_1 - x_2), e_2 = (x_2 - x_3), e_3 = (x_3 - x_1)$$

Figure 3.2 Minimum incidence angle constraint



The distance from the triangle is expressed as

$$d = (g - x_1)^T a_N$$

The constraints given by minimum and maximum distance is written as

$$d_{max} \geq (g - x_1)^T a_N \geq d_{min}$$

Combining distance and minimum incidence angle constraints.

$$\begin{bmatrix} (g-x_i)^T n_i \\ (g-x_1)^T a_N \\ -(g-x_1)^T a_N \end{bmatrix} \geq \begin{bmatrix} 0 \\ d_{min} \\ -d_{max} \end{bmatrix}, i = 1,2,3$$

Position constraints dependent on craft orientation

The cameras field of view is given by horizontal ψ_{hor} and vertical spread ψ_{ver}. The field of view is rotated about the y-axis by the assumed constant pitch. Because of the constant pitch, positions g where the entire triangle is visible for some rotation about the z axis is not convex.

Figure 3.3 Left and right normals used for convexification, by rotating feasible region around the center of the triangle

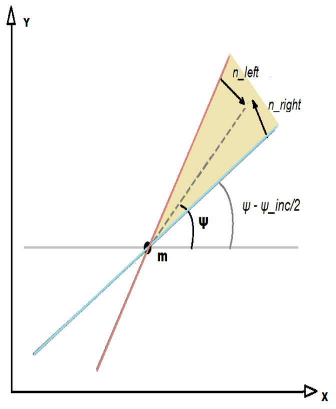
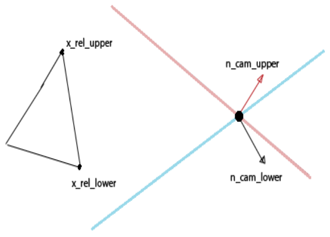


Figure 3.4 Top and bottom camera bound, and direction of normals



By dividing the $xyplane$ into N sections defined by hyperplanes with normals n_{right}, n_{left} as seen in figure 3.3. The rotation of the craft is assumed to be rotating along with the hyperplanes. The nonconvex problem is solved as N separate approximately convex problems, the local solutions are compared to select the optimal position. Given x^{rel}_{lower} and x^{rel}_{upper} as the respective relative corners of the mesh triangle that are the corners of the triangle constraining choice of g the most. m is the triangle center and $n^{cam}_{lower}, n^{cam}_{upper}$ are rotated camera normals. And rotation increment from section to section as $\psi_{incr} = 2\pi/N$

$$\phi_{lower} = \phi_{cam} + \theta_{ver}/2 \ , \ \phi_{upper} = \phi_{cam} - \theta_{ver}/2$$

$$n_{right} = \begin{bmatrix} -\sin(\psi_i - \psi_{incr}/2.0) \\ \cos(\psi_i - \psi_{incr}/2.0) \\ 0 \end{bmatrix} n_{left} = \begin{bmatrix} -\sin(\psi_i + \psi_{incr}/2.0) \\ \cos(\psi_i + \psi_{incr}/2.0) \\ 0 \end{bmatrix}$$

$$n^{cam}_{lower} = \begin{bmatrix} \sin\phi_{lower}\cos\psi_i \\ \sin\phi_{lower}\sin\psi_i \\ -\cos\phi_{lower} \end{bmatrix} n^{cam}_{upper} = \begin{bmatrix} \sin\phi_{upper}\cos\psi_i \\ \sin\phi_{upper}\sin\psi_i \\ -\cos\phi_{upper} \end{bmatrix}$$

$$x^{rel}_{lower} = \max_{x_i} x_i^T n^{cam}_{lower} \ , \ x^{rel}_{upper} = \max_{x_i} x_i^T n^{cam}_{upper}$$

Horizontal angular constraints are not encoded but are handled by choosing d_{min} large enough to guarantee to observe the entire triangle.

$$\begin{bmatrix} (g - x^{rel}_{lower})^T n^{cam}_{lower} \\ (g - x^{rel}_{upper})^T n^{cam}_{lower} \\ (g - m)^T n_{right} \\ (g - m)^T n_{left} \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Let g^{k-1} be viewpoint sampled for observing the same triangle the previous algorithm iteration. g^{k-1}_p the preceding viewpoint on the old tour and g^{k-1}_s the subsequent viewpoint on the old tour. The optimization goal is to minimize the distance traveled by the craft by decreasing the distance between the current iteration viewpoint and its subsequent viewpoint, preceding viewpoint and the viewpoint observing the same triangle of the best tour generated so far.

This results in a covex QP with linear constraints, meaning it can be solved by efficient solvers such as the active set solver by Ferreau et al. (2014).

$$\min_{g^k} (g^k - g^{k-1}_p)^T (g^k - g^{k-1}_p) + (g^k - g^{k-1}_s)^T (g^k - g^{k-1}_s) + (g^k - g^{k-1})^T (g^k - g^{k-1})$$

$$st. \begin{bmatrix} n_1^T \\ n_2^T \\ n_3^T \\ a_N^T \\ -a_N^T \\ n^{cam}_{lower} \\ n^{cam}_{upper} \\ n_{right} \\ n_{left} \end{bmatrix} g^k \geq \begin{bmatrix} n_1^T x_1 \\ n_2^T x_2 \\ n_3^T x_3 \\ a_N^T x_1 + d_{min} \\ -a_N^T x_1 - d_{max} \\ n^{cam}_{lower} x^{rel}_{lower} \\ n^{cam}_{upper} x^{rel}_{upper} \\ n_{right} m \\ n_{left} m \end{bmatrix}$$

The viewpoint is further constrained to be within the bounding box specified as a parameter. $g = [x, y, z]$

$$X_{min} < x < X_{max}, Y_{min} < y < Y_{max}, Z_{min} < z < Z_{max}$$

Heading

Heading is determined with a grid search, and is computed given the cost function:

$$cost = (\alpha_p^{k-1} - \psi^k)^2/d_p + (\alpha_s^{k-1} - \psi^k)^2/d_s$$

$$\alpha_p^{k-1} = \tan(\frac{y_k - y_p^{k-1}}{x^k - x_p^{k-1}}), \alpha_s^{k-1} = \tan(\frac{y_s^{k-1} - y^k}{x_s^{k-1} - x^k})$$

Minimizing the change in yaw from the heading of the copter, given the constraints that the triangle should be fully visible.
Given constraints on minimum and maximum distance, as well as minimum incidence angle 3.1, 3.2 , the triangle is evaluated as visible if.

$$\begin{bmatrix} a_N^T(g^k - x_1 - d_{min}a_N) \\ -a_N^T(g^k - x_1 - d_{max}a_N) \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} (g^k - x_1)^T n_1 \\ (g^k - x_1)^T n_2 \\ (g^k - x_1)^T n_3 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

The field of vision is represented by four normals bounding from left, right, below and above. The normals are given by the fixed camera pitch ψ_{cam}, horizontal spread θ_{hor} and vertical spread θ_{ver}.

$$c_{N1} = \begin{bmatrix} -sin(\phi_{cam} - \theta_{ver}/2) \\ 0 \\ -cos(\phi_{cam} - \theta_{ver}/2) \end{bmatrix} c_{N2} = \begin{bmatrix} sin(\phi_{cam} + \theta_{ver}/2) \\ 0 \\ cos(\phi_{cam} + \theta_{vert}/2) \end{bmatrix}$$

$$c_{N3} = R_z(\theta_{hor}/2) \begin{bmatrix} sin(\phi_{cam}) \\ 0 \\ cos(\phi_{cam}) \end{bmatrix} c_{N4} = R_z(-\theta_{hor}/2) \begin{bmatrix} sin(\phi_{cam}) \\ 0 \\ cos(\phi_{cam}) \end{bmatrix}$$

The triangle is evaluated as visible if.

$$R_z(\psi)R_y(\phi)c_{Nj}^T(x_i - g^k) \geq 0, i = 1, 2, 3, j = 1, 2, 3, 4$$

where ψ is yaw and ϕ is pitch. If all six inequalities hold, then each point x_i is within the bounding hyperplanes defined by the camera normals c_{Nj} and position g. This holds for $\theta_{hor}, \theta_{ver}$ less than 180^o.

The final step of the visibility criterion is to perform a collision check between the state and the triangle, however that is not covered in this paper.

Initial values

Initial positions are placed in the least constrained position for each triangle, without optimizing the distance between viewpoints. To find the initial heading the view is simply rotated by the angle increment until the entire triangle is visible.

3.1.3 Tour length estimation

The estimated time to complete a tour is given as the sum of time it takes to transition from one viewpoint to another. With linear movement and rotation, with bounding translational velocity v_{max} and bounding angular velocity ω_{max}, given N viewpoints and viewpoint g_0, g_{N+1} being the requested start and goal positions of the tour.

$$cost = \sum_{k=1}^{N+1} \max(\frac{\sqrt{(g_k - g_{k-1})^T((g_k - g_{k-1})}}{v_{max}}, c(\psi_k, \psi_{k-1}, \omega_{max}))$$

Where $c(\theta_k, \theta_{k-1}, \omega)$ is the smallest amount of time needed to rotate from one angle to another given $\theta = [-\pi, \pi]$.

$$c(\theta_k, \theta_{k-1}, \omega_{max}) = \frac{mod((\theta_k - \theta_{k-1}), \pi)}{\omega_{max}}$$

Adding pitch to the equation, assuming the manipulator or craft still has the given angular and translational velocity bounds.

$$cost = \sum_{k=1}^{N+1} \max(\frac{\sqrt{(g_k - g_{k-1})^T((g_k - g_{k-1})}}{v_{max}}, c(\psi_k, \psi_{k-1}, \omega_{max}), c(\phi_k, \phi_{k-1}, \omega_{max}))$$

3.1.4 Path Smoothing

Due to the mobility limitations, the planning algorithm will attempt extra measures to smooth the resulting trajectory when planning for fixed-wing AUVs. When preceding state and succeeding state are determined before used to sample a new viewpoint. The smoothing algorithm will attempt to instead choose states up to w_{max} viewpoints away on the tour. w_{max} is calculated by.

$$p_{smooth} = I_{max}/2.25 \quad n_{steps} = \min(2.0, 1.0 + 50.0/I_{max})$$
$$w_{max} = \min(n_{steps}(I_{max} - p_{smooth} - i), N/2.0)$$

Where I_{max} is the maximum number of iterations i is the current iteration and N is the total number of triangles making up the mesh.

$s_1, \quad s_2$ are the preceding and succeeding states, *tour* is an ordered list of waypoints defining the inspection tour. *tri* is a vector of all triangles forming the mesh, ID is the current triangle, and the lastVisible function performs an RRT* expansion connecting the given state the expansion its called from to the state given as an argument and returns the state closest to the state the expansion is called from where the triangle was visible. Thus

Algorithm 2 RRT*

```
1: $s_1 \leftarrow precedingviewpoint$
2: $lim \leftarrow 0$
3: while $isDefined(s_1) \land lim.increment() \leq w_{max}$ do
4:     $s_1 \leftarrow tour(current - lim).lastVisible(s_2, tri[ID])$
5: $s_2 \leftarrow succeedingviewpoint$
6: $lim \leftarrow 0$
7: while $isDefined(2_1) \land lim.increment() \leq w_{max}$ do
8:     $s_2 \leftarrow tour(current + lim).lastVisible(s_2, tri[ID])$
```

this should move the preceding and succeeding states further along the tour until they are on the edge of being able to observe the triangle.

The *isDefined* function represents checking for if the coordinates of the state vectors are real numbers.

The intention is to cause optimal viewpoints to align on linear or curving paths instead of simply contracting towards one another.

3.1.5 Previous expansions on the structural inspection planner

For a manipulator there is little sense to restrict pitch of the endeffector to a predetermined constant value. Therefore the first expansion was to increase degrees of freedom from 4 to 5 by introducing variable pitch. This section will elaborate changes done in Seines (2016), as such this section is directly copied from that paper.

Relaxing vertical camera constraints

With a locked pitch a great deal of the issue is finding a convex space the copter can be in and be able to observe the entire triangle. However with the introduction of pitch the space with any valid orientation is greatly increased. 3.1

The constraints given by minimum incidence form a tetrahedron shaped feasible region when constrained by maximum distance. Assume some fixed camera spread in both horizontal and vertical direction $\theta_{hor}, \theta_{ver}$. Splitting the grid search into two 2D problems, for there to not exist a feasible heading observing the triangle then the camera must be in a position g such that the angle between the vectors pointing towards the extreme points on the triangle in either direction $(g - x^{rel}_{lower}), (g - x^{rel}_{upper})$ and $(g - x^{r}_{right}el), (g - x^{rel}_{left})$ is larger than $min(\theta_{hor}, \theta_{ver})$. If the minimal incidence angle constraint is satisfied this angle is at most equal to $\pi - 2\theta_{min}$ where these hyperplanes intersect, and is decreasing moving along the boundaries of either constraint and with increasing distance from the triangle. Thus selecting θ_{min} such that;

$$\theta_{min} >= \pi/2 - min(\theta_{hor}, \theta_{ver})/2$$

Implies any position stratifying the constraints given by minimum incidence angle will have at least one feasible heading for inspecting the whole triangle.

The new optimization constraints are then given as.

$$min_{g^k}\ (g^k - g_p^{k-1})^T(g^k - g_p^{k-1}) + (g^k - g_s^{k-1})^T(g^k - g_s^{k-1}) + (g^k - g^{k-1})^T(g^k - g^{k-1})$$

$$st. \begin{bmatrix} n_1^T \\ n_2^T \\ n_3^T \\ a_N^T \\ -a_N^T \end{bmatrix} g^k >= \begin{bmatrix} n_1^T x_1 \\ n_2^T x_2 \\ n_3^T x_3 \\ a_N^T x_1 + d_{min*} \\ -a_N^T x_1 - d_{max} \end{bmatrix}$$

Where n_1, n_2, n_3 are calculated as above with a θ_{min} stratifying the relation with $min(\theta_{hor}, \theta_{ver})$.

Optimalization criterion for heading

Since manipulator movement is suitable for having the sensor move perpendicular along the object surface, resulting in a bigger portion of the object is within the FOV at all times, compared to a sensor heading in the direction of horizontal movement whenever the sensor moves. The normalized vector pointing towards the focus of the sensor is given by the vector.

$$c_f = \begin{bmatrix} cos(\phi)cos(\psi) \\ cos(\phi)sin(\psi) \\ sin(\phi) \end{bmatrix}$$

Let d_p, d_s be distances between this viewpoint and subsequent and preceding viewpoints. Then the proposed cost function is given by.

$$(1+c_f(\psi^k, \phi^k)^T a_N) + \frac{\sqrt{(\psi_p^{k-1} - \psi^k)^2 + (\phi_p^{k-1} - \phi^k)^2)}}{d_p} + \frac{\sqrt{(\psi_s^{k-1} - \psi^k)^2 + (\phi_s^{k-1} - \phi^k)^2)}}{d_s}$$

Along with a constraint minimizing change in yaw and pitch from one state to the next with the goal of enforcing smooth camera movement.

3.1.6 New expansions of the inspection planner

Further expansion focus on adapting the algorithms to take account for manipulator kinematics. Such as avoiding viewpoints with no viable solutions, or infinite solutions from inverse kinematics.

To penalize connections between viewpoints where the manipulator requires arcs to avoid obstacles or self-collisions, the distance evaluation function has been expanded to include a simple collision check along the path generated by linearly interpolating between viewpoints and solving for joint states along the path.

Inverse kinematics viability criterion

Singularities and poses prone to self-collision must be taken into account while sampling an optimal viewpoint, the presence of inverse kinematics solution as well as if the solution is in a self-collision is checked. If the checks fail, the algorithm will search for random valid states within a small distance of the original viewpoint. Inspired by the redundant roadmap planning approach, random configurations are sampled until K valid states are found observing the viewpoint. Returning fewer configurations the number of attempts exceeds a given limit. The optimal solution of the K alternatives is calculated based on the same objective functions as specified in the sections above. If no solution is found the point is discarded as unfeasible and treated as if it is in collision.

If a valid point is found, then whether or not to add the point to the tour is computed as normal for the new point. Algorithm presented below 3. r_L is a randomly generated vector of fixed length. During simulations K have been between 1 and 5, and K_it has been between 20 and 100.

Algorithm 3 Redundant roadmap inspired point viability criterion

```
1:  Iterate viewpoint configuration
2:  Compute inverse kinematics
3:  Check collisions and configuration validity.
4:  if No solution or invalid state then
5:      $solFoundLocal \leftarrow false$
6:      $sol\_ vector \leftarrow empty\ vector$
7:      $k \leftarrow 0 \quad or \quad it \leftarrow 0$
8:      while $k < K \quad it < K_{it}$ do
9:          $g_r^k = g^k + r_l$
10:         if $g_r^k$ is valid then
11:             $sol\_vector \leftarrow g_r^k$
12:             $k \leftarrow k + 1.$
13:         $it \leftarrow it + 1$
14:     $c_{wp} \leftarrow \inf$
15:     for $g_r \in sol\_ vector$ do
16:         $c_{tot} \leftarrow calcObjectiveFunc(g_r) + calcRotationCost(g_r)$
17:         if $c_{tot} < c_{wp}$ then
18:             $g \leftarrow g_r \quad c_{wp} \leftarrow c_{tot}$
19:             $solFoundLocal \leftarrow true$
20: if $solFoundLocal \wedge c_{wp} < c_{prev}$ then
21:     $best \leftarrow g$
```

This is an effort to minimize the number of invalid points, and to ensure all points have at least one valid inverse kinematics solution without collision.

Augmented distance evaluation function

For motion planners planning linear paths in tool space, linear constant velocity movement holds, assuming a straight path is feasible between the points. And the norm 2 distance between the points will be descriptive of the time required to move between them. However planning in joint space with the planners in the OMPL this is not a guarantee. However, the manipulator will approximate this behavior for sufficiently close points or if the linear path between the viewpoints is collision and singularity free.

The linear path between points is interpolated and the presence of inverse kinematics solutions, as well as validity, is checked along the way. Since most of the planners will handle singularities and avoid self-collisions, the percentage of iterations where this is present along the linear path is correlated with likelihood of arching motions deviating from a linear path and is added to the weight as a penalty. Invalid start or end state is treated as if the viewpoint is in a collision.

The intention is to increase the likelihood of a path where the order of the viewpoint matches ideal movement for a manipulator. However, it does not prevent the occurrences of paths containing several arching motions from point to point. The extension for the

Algorithm 4 linear path viability check extension

```
1: $dist \leftarrow computeDistance(s_1, s_2)$
2: $n_{error} \leftarrow 0$
3: $s_{ik1} \leftarrow inverse(s_1) \quad s_{ik2} \leftarrow inverse(s_2)$
4: if $checkCollision(s_{ik2}) \lor !isFeasible(s_{ik2} \lor checkCollision(s_{ik1}) \lor !isFeasible(s_{ik1})$
   then
5:     $collision = true$
6: for $it \in [0, 0 + g_{discStep}, ..1]$ do
7:     $s \leftarrow (its_1 + (1 - it)s_2)$
8:     $s_{ik} \leftarrow inverse(s)$
9:     if $checkCollision(s_{ik}) \lor !isFeasible(s_{ik})$ then
10:        $n_{error} \leftarrow n_{error} + 1$
11: $c_{penalty} \leftarrow C_{scale} n_{error}$
12: $dist_{ex} = dist + c_{penalty}$
```

algorithm calculating the distance between two viewpoints $s_2 \quad s_1$. *inverse* is a function returning joint configurations putting the end effector in the given pose. *checkCollision* checks self collision and collision with obstacles for all robot joints for a given joint state. *isFeasible* is a function that checks if returned values are within joint limits. C_{scale} is a variable depending on $||s_1 - s_2||$ to compensate for the fact that the path is divided into $1/g_{discStep}$ regardless of lenght between the points.

3.1.7 Point to point planning

The structural inspection planner plans in tool space, with a point to point planning algorithm doing simple linear interpolation short distances and relying on RRT* for planning longer paths. Collisions are checked according to the discretization parameter $g\_discretization\_step$. If a collision is found while interpolating the RRT* algorithm is called to generate a collision-free path.

As a result, the output trajectory of the system is in tool space by default. However, this can still be used to move a manipulator by using the waypoints as tool space goals, for a multigoal planning algorithm.

RRT*

The RRT* algorithm is an asymptotically optimal path planning algorithm proposed by Karaman and Frazzoli (2011). The algorithm is based on the data structure rapidly expanding random tree(RRT), after which it is named. The algorithm will result in a path if a given node is within a given distance of the goal distance, however, the algorithm will generally run for a given number of iterations.

The algorithm adds new states to the tree to the existing branch such that the edges of the graph are all part of an optimal path from the start state to one of the nodes. Whenever a new state is added, if it provides a new optimal path to any vertex near it, the tree is updated by removing the connection between the vertex and the previously optimal parent.

Algorithm 5 RRT*

```
1: $V \leftarrow x_{start} \quad E \leftarrow \emptyset$
2: **while** $n < N_{plan}$ **do**
3:     $n \leftarrow n + 1$
4:     $x_{rand} \leftarrow$ random valid state
5:     $x_{nearest} \leftarrow$ nearest existing node
6:     $x_{opt} \leftarrow$ solution of: $\min_{x_{opt}} ||x_{opt} - x_{nearest}||$ st. $||x_{rand} - x_{opt}|| \leq \mu$
7:     **if** $obstacleFree(connect(x_{near}, x_{opt}))$ **then**
8:         $X_{near} \leftarrow$ all nodes within a given radius.
9:         $V \leftarrow x_{opt}$
10:        $x_{min} \leftarrow x_{nearest}, c_{min} \leftarrow cost(c_{nearest}) + cost(Line(x_{nearest}, x_{opt}))$
11:        **for** each $x_{near} \in X_{near}$ **do**
12:            $c_{near} \leftarrow cost(x_{near}) + cost(connect(x_{opt}, x_{near})$
13:            **if** $c_{near} < c_{min} \wedge collisionFree(connect(x_{opt}, x_{near}))$ **then**
14:                $c_{min} \leftarrow c_{near} \quad x_{min} \leftarrow x_{near}$
15:        $E \leftarrow E \cup connect(x_{min}, x_{opt})$
16:        **for** each $x_{near} \in X_{near}$ **do**
17:            update the tree if the new minimal
18:            **if** $cost(x_{opt}) + cost(Line(x_{near}, x_{opt})) < cost(x_{near}) \wedge collisionFree(connect(x_{opt}, x_{near}))$ **then**
19:                $x_{parent} \leftarrow parent(x_{near})$
20:                $E \leftarrow (E \setminus Path(x_{parent}, x_{near})) \quad E \cap connect(x_{new}, x_{near})$
```

V is a data structure containing all nodes and E is an ordered graph containing all edges between nodes. *Connect* is a function returning the value of an edge connecting two nodes. *Path* returns the edges connecting two nodes. *Line* returns length of a straight line connecting the two nodes.

Since the algorithm is general purpose, it should be possible to adapt it to plan paths in joint space.

3.2 Moving the robot

Since the algorithm by default returns a tool space trajectory, specified by a number of waypoints with position and orientation, this needs to be translated into a joint space trajectory to move the robot.

There are multiple different ways to make a UR5 execute a specified trajectory.

3.2.1 UR joint trajectory interfaces

Robots from Universal Robots have three different controller APIs. First, the graphical user interface level for programming and control with the teach pendant. Secondly, there is the script level API where the robot is passed scripts in a robot programming language and the robot control box execute these scripts. The third is the C-API level where control is done in the forms of executable programs passing trajectory messages through sockets.

The move group node is a wrapper that handles robot control for the user, implemented in the C-API. The move group node accepts trajectory requests in the form of a robot trajectory message object.

```
message moveit_msgs/RobotTrajectory.msg
    trajectory_msgs/JointTrajectory joint_trajectory
    trajectory_msgs/MultiDOFJointTrajectory
       multi_dof_joint_trajectory
```

```
message trajectory_msgs/JointTrajectory.msg
    std_msgs/Header header
    string[] joint_names
    trajectory_msgs/JointTrajectoryPoint[] points
```

```
message trajectory_msgs/JointTrajectoryPoint.msg
    float64[] positions
    float64[] velocities
    float64[] accelerations
    float64[] effort
    duration time_from_start
```

The robot trajectory message specifies trajectories in the form of multiple points. If acceleration, velocities, effort or position stray from the reference the robot controller will interpret this as a path constraint violation if the error crosses a given threshold, and trajectory execution is canceled.

The robot can be controlled by passing trajectory messages to the move group node, that forward the desired trajectory to the robot driver. Both the tool space motion planner and the joint space point to point motion planner uses this control API by calling functions defined in MoveIt libraries to generate trajectory messages and execute them.

URscript

An alternate control method is through URscript `http://www.sysaxes.com/manuels/scriptmanual_en_3.1.pdf`. URscript is the robot programming language of universal robots and is used for the script level API. To use this approach, the program generates a script defining the requested movement, streams it to the robot control box which executes the script.

URscript has functions for moving from one pose to another linearly in tool space, moving from one configuration to another linearly in joint space as well as circular motion and online control of the robot.

3.2.2 OMPL and ROS

ROS and moveIt nicely provide both collision avoidance and motion planning for kinematic chains/manipulators. Serves as the basis for several different approaches to path planning.

Motion planning in tool space

Planning in tool space is the simplest approach with minimal specificity, the path is defined by one or more goal poses and MoveIt handles solving inverse kinematics for each pose and connecting poses, as well as collision avoidance and local point to point planning. The move group interface provides functionality for single goal planning, with the focus on a safe and collision-free execution.

Pseudo code for a simple move

Algorithm 6 Tool space point to point movement

```
1: initialize ROS node
2: $move\_group \leftarrow initializeMoveGroupInterface(manipulator)$
3: $PublishCollisionObjects()$
4: $waypoints \leftarrow loadWayPointsFromFile(filename)$
5: for $eachwaypoint \in waypoints$ do
6:     $move\_group.poseTarget \leftarrow waypoint$
7:     $move_group.move()$
```

As well as functionality to compute a plan in tool space that takes multiple poses as input. The tool space planner works by specifying a crawling distance to approximate linear movement of the end effector between poses by interpolating and computing inverse kinematics along the path.

Local point to point planning is handled by the default planning plugin provided by OMPL, usually single query random planner such as RRTConnect.

The tool space path computation can be defined by the following pseudo code. full code in the appendix.

The Cartesian motion planning function iterates along a linear path given a fixed step size while computing inverse kinematics and checking for a collision at each step.

Algorithm 7 Cartesian point to point movement

```
1: initialize ROS node
2: $move\_group \leftarrow initializeMoveGroupInterface(manipulator)$
3: $PublishCollisionObjects()$
4: $waypoints \leftarrow loadWayPointsFromFile(filename)$
5: $Trajectory \leftarrow emptyRobotTrajectory()$
6: $Trajectory \leftarrow move\_group.ComputeCartesianPath(waypoints, max\_step\_size, jump\_treshold)$
7: $move_group.execute(Trajectory)$
```

There is a more specific package made for use with ROS for this purpose. Descartes is a planning package providing more advanced functionality regarding planning paths in tool space. Providing its own core functionality based on the functionality provided by moveIt. As well as a dense planner, planning optimized path for multiple goals in tool space.

Motion planning in joint space

The slightly more low level approach involves computing inverse kinematics, planning in joint space with joint limits to enforce desired behavior. In this case, ROS is still handling collision avoidance, as well as point to point planning. However by requesting given configurations instead of poses, the state of the robot as it reaches each goal state is deterministic.

A common approach to path planning is randomly expanding the state while checking the validity of the expansions until the goal state is reached. The path is then smoothed and optimized through post processing. While this is often a quick way of implementing point to point planning, it may in some cases result in obviously non-optimal paths. As minimizing time spent planning is often the focus of these kinds of planners.

An alternative approach is to also do both collision avoidance, velocity, and acceleration computations. To create the trajectory yourself and stream it to the robot using either the simple action interface, publishing the trajectory directly to the robot or use UR script. To avoid usage of the move group node.

3.3 Point cloud acquisition

As the focus of the paper is inspection and motion planning, the selected approach to point cloud reconstruction is a simple greedy algorithm, reconstructing a triangle based mesh from one or more point clouds put together.

To reconstruct the entire object is it is desirable to transform all point clouds from the sensor frame to a fixed coordinate system.

3.3.1 Representing measured point cloud data in a fixed frame

The end effector position and orientation can be calculated from joint states by forward kinematics. Since the sensor is attached to the end effector the sensor position and orien-

Algorithm 8 Tool space point to point movement

```
1: initialize ROS node
2: $robot\_model \leftarrow loadRobotModel(modelSpecifier)$
3: $planner \leftarrow loadPlanner(OMPL/PlannerOfChoice)$
4: $scene \leftarrow addRobot(robot\_model)$
5: $scene \leftarrow addCollisionObjects(objects)$
6: $move\_group \leftarrow initializeMoveGroupInterface(manipulator)$
7: $waypoints \leftarrow loadWayPointsFromFile(filename)$
8: for $each \quad waypoint \in waypoints$ do
9:     $current\_state \leftarrow move\_group.getCurrentState()$
10:    $ik\_solutions \leftarrow inverse(waypoint)$
11:    for $each\ config \in ik\_solutions$ do
12:        if $isValid(config) \wedge !checkCollision(config)$ then
13:            $valid\_solutions \leftarrow config$
14:    $d_{min} \leftarrow \inf$
15:    $c_{opt} \leftarrow -1$
16:    for $config \in valid\_solutions$ do
17:        if $weightedDistance(config, current\_state) \leq d_{min})$ then
18:            $d_{min} \leftarrow weightedDistance(config, current\_state) \quad c_{opt} \leftarrow config$
19:    $textitplanning\_request \leftarrow workspaceBounds(), current\_state, c_{opt}$
20:    $plan \leftarrow planner.plan(planning\_request)$
21:    $move\_group.execute(plan)$
```

tation is logged indirectly through logging the robot joint states.

The point cloud is given as positions in x,y,z coordinates relative to the camera. Using the fixed world frame of the robot as the default frame for the point clouds a transformation from the camera frame to the world frame is expressed by.

$$R = forwardKinematics(\textit{joint\_states})$$

$$PC_{world} = R^{-}1PC_{sensor}$$

Where PC_{sensor} is the point cloud given in the sensor frame and PC_{world} is the point cloud in the world frame.

3.3.2 Making a 3D mesh

Marton et al. (2009) proposes a greedy algorithm for generating a mesh from noisy point cloud data sets of either one point cloud or multiple point clouds put together.

Algorithm 9 Greedy algorithm from mesh generation from pointcloud file

```
1: $\textit{blob} \leftarrow loadPointCloud(\textit{filepath})$
2: $\textit{normals} \leftarrow cumputeNormals(\textit{blob}$
3: $\textit{point\_cloud} \leftarrow concatenateFields(\textit{blob,} \quad \textit{normals})$
4: $\textit{search\_tree} \leftarrow initialize(\textit{point\_cloud})$
5: $\textit{mesh} \leftarrow emptyMesh()$
6: $\textit{gp3} \leftarrow initializeGreedyProjectionTriangulation()$
7: $\textit{gp3}.setInputData(\textit{point\_cloud})$
8: $\textit{gp3}.setSearchMethod(\textit{search\_tree})$
9: $\textit{mesh} \leftarrow \textit{gp3}.reconstruct()$
```

Where the gp3 object represents a greedy triangulation algorithm, locally searching the search tree for a given number of nearest neighbors and search radius to connect pairs of three points into triangles. Surface normals are used to ensure object shape is preserved, by connecting triangles such that the surface normal of the given triangle is equal to the normal of the estimated normal of the surface formed by the points.

Chapter 4

Robot setup

4.1 Dependencies

The packages that needs to be installed to follow the setup presented in this chapter. Top cell contains the dependencies for the structural inspection planner. The second cell contains the dependencies of robot path planning and simulations. The third cell contains dependencies for running the Intel realsense sr300, as well as to perform operations on pointclouds. The bottom row is the package used to auto generate configuration files for the robot.

Packages without a provided url can be installed with apt-get provided apt has been given the official ros package list. The packages with a given url has to be buildt from source.

Dependencies
libeigen3-dev ros-indigo-tf ros-indigo-rviz ros-indogo-octomap ros-indigo-octomap-msgs
ros-indigo-moveit ros-indigo-universal-robot ros-indigo-ur-kinematics ros-indigo-trac-ik descartes (`https://github.com/ros-industrial-consortium/descartes`)
ros-indigo-realsense-camera librealsense (`https://github.com/IntelRealSense/librealsense`) ros-indigo-pcl-ros
ros-indigo-moveit-setup-assistant

4.2 Making endeffector and configuring robot

As previously mentioned, the inspection planner, the simulations, and the driver for the 3D camera, is implemented as packages for Robot Operating System indigo(ROS indigo).

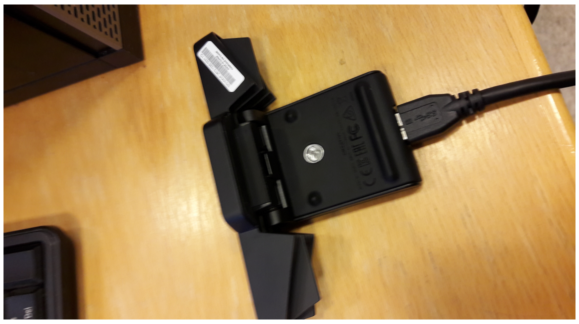
To inspect the robot the SR300 was attached as a tool to the third wrist joint of the robot. To avoid collision between the sensor and the robot link the robot description files are modified to include collision information for the sensor, and to allow kinematic plugins to perform forward kinematics and inverse kinematics between joint space and the sensors position in tool space.

Attaching SR300

First to attach the sr300 to the robot I had to make a mount for it, that would fit the holes on the end link of the ur5.

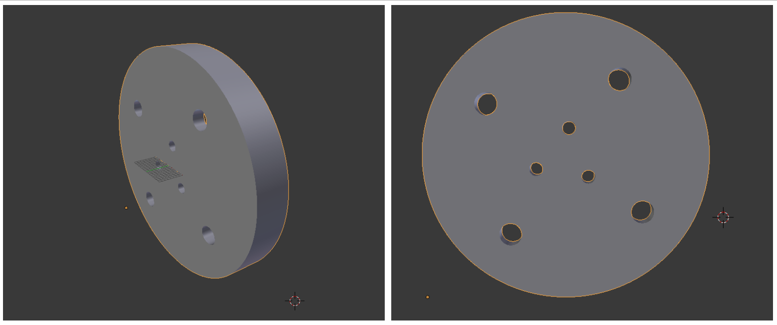
To fasten the camera to the ur5 steadily the idea was to make a mount where the camera is fastened tightly by fastening a standard camera screw (shown in figure 4.1).

Figure 4.1 The sr300 comes attached to a plate with the usb outlet as well as a hole for a standard camera mounting screw



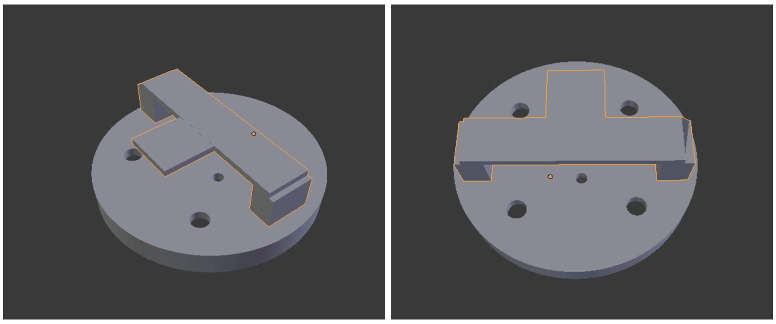
To attach a custom mount to the ur5 first a base is needed to attach to the wrist link. To this end SINTEF had a model of a cap that would fit the end effector forming a nice base. The endeffector is made by combining this with a mount for the camera.

Figure 4.2 STL model for 3D printed base, to fit with ur5 end effector base.



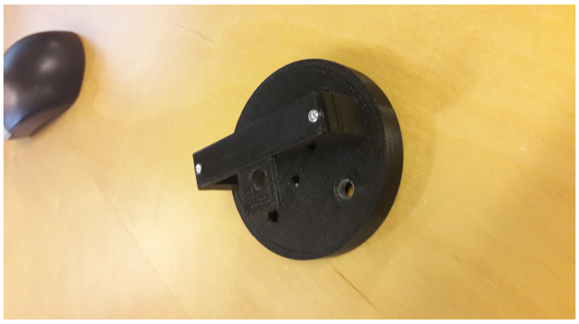
The sr300 camera only bends towards the underside of the plate, to allow the sensor to point out from the wrist link the mount has to wrap around the sensor. The base is fastened to the ur5 with bolts not embedded in the base, as such there need to be some extra height under the mount. 3D printed material is easy to shape with drills or knifes after printing, as such the camera mount was made with the intention of drilling a holes to fasten the mount with screws.

Figure 4.3 Intended end effector before mounting camera, holes for screws for attaching mount to base and mount to camera were drilled after printing



Results after fastening the mount to the base, and drilling a hole for attaching the sensor.

Figure 4.4 Completed camera mount



Measurements on the end effector places the center of the camera box approximately 4.5 cm out and 3.5 cm up from the end effector link.

$$ee_{offset} = \begin{bmatrix} 0.045 \\ 0.0 \\ 0.035 \end{bmatrix} m$$

The sensor has a height and depth of approximately 3 cm, and a width of 12 cm. For

Figure 4.5 End effector put together, and attached with the camera to the robot

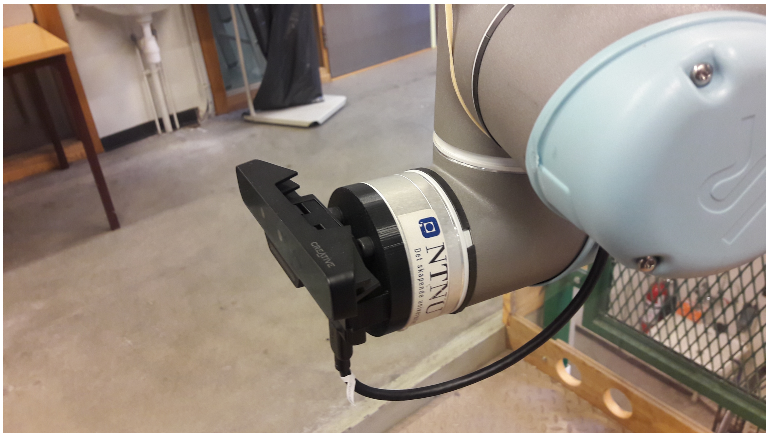
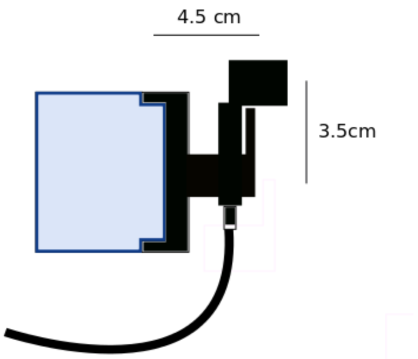


Figure 4.6 End effector attachment measurements



4.2.1 Configuring URDF, SRDF and kinematics plugins for custom robot

Unified Robot Description Format or URDF for short, is a description format for describing a robot. An URDF describing a robot consists of a robot definition, where links and joints rotation, visuals and collision matrices are described. As well as the order they are coupled together.

For ROS to be able to perform collision checks and path planning for the robot it needs to know what it looks like, how it behaves and have the shape of its links defined by a collision matrix.

Semantic Robot Description Format or SRDF for short, is a format for describing robot semantics such as groups of joints intended to be moved together and define the name and link of the endeffectors for a given joint group. The SRDF also specifies adjacent links and links that cannot collide such that collision checks can ignore collision between the specified links.

Since scripts used to launch nodes and find configuration files, find the required files with the help of ROS package navigation it is recommended to follow a file structure similar to this: `https://github.com/mmseines/ur5_simulation/tree/master/src/ur5_robot`.

Configuring URDF

By installing the universal robots package it will install configuration files for all UR robots without tools attached.

These can be found in the package ur_ description. To attach the sensor a new xacro file is made by copying the existing ur5_ robot.urdf.xacro file. It is recommended that the urdf file is compiled as an appropriately named package, and linked with the shared ros files. Such that the file can be found from any other file by searching for the package name. For instructions for making a catkin package consult the tutorials found on the ROS home page.

Additional joints and links are added below. Declaration of sensor joint and link shown below. Full URDF xacro in appendix section 7.4.4.

```

<! -- visualize the sensor -->
  <link name="sensor">
    <visual>
        <origin xyz="0.0 0.0 0.0" rpy="0 0.0 0.0" />
        <geometry>
            <box size="0.03 0.12 0.03"/>
        </geometry>
        <material name="Grey" />
    </visual>
    <collision>
        <origin xyz="0.0 0.0 0.0" rpy="0 0 0" />
        <geometry>
            <box size="0.03 0.12 0.03"/>
        </geometry>
    </collision>
  </link>

  <! -- Specify joint representing origin of sensor -->
  <joint name="tool_to_sensor" type="fixed">
    <parent link="ee_link"/>
    <child link="sensor"/>
    <origin xyz="0.045 0.0 0.035" rpy="0 0.0 0.0" /> <!--
       THIS LINE WAS MISPLACED DURING EXPERIMENTS -->
  </joint>
```

An urdf file can be generated by opening a terminal and running the commend.

- rosrun xacro xacro.py custom_ ur5.urdf.xacro > custom_ ur5.urdf

Generating config files and SRDF

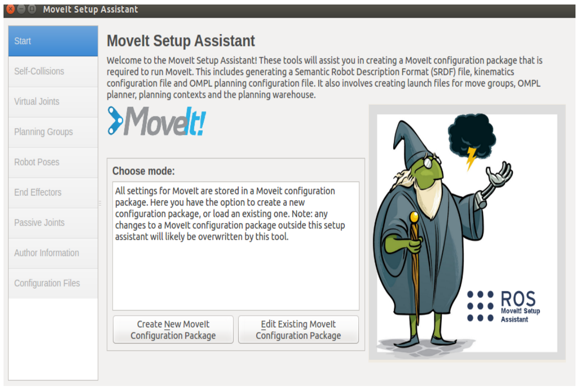
Writing configuration files by hand would be impractical, therefore it is recommended generating most of the configuration files using the setup assistant.

After customizing the URDF, open a terminal and run the setup assistant with the command.

- roslaunch moveit_ setup_ assistant setup_ assistant.launch

This will open this window. Since the moveit configurations for the default UR robots is located in the ros folder in /opt, it is not reccomended to alter them. Instead, make a new configuration package located in the planner project workspace.

Figure 4.7 Moveit setup assistant



MoveIt Setup Assistant

Welcome to the MoveIt Setup Assistant! These tools will assist you in creating a MoveIt configuration package that is required to run MoveIt. This includes generating a Semantic Robot Description Format (SRDF) file, kinematics configuration file and OMPL planning configuration file. It also involves creating launch files for move groups, OMPL planner, planning contexts and the planning warehouse.

Choose mode:

All settings for MoveIt are stored in a Moveit configuration package. Here you have the option to create a new configuration package, or load an existing one. Note: any changes to a MoveIt configuration package outside this setup assistant will likely be overwritten by this tool.

Upen the custom_ robot.urdf.xacro file and follow the steps one by one, be sure to select the sensor joint.

Define a group to plan for and the joints the group consists of.

Define kinematics solver. for this project the kinematics plugin of choice will be the analytic solver by Hawkins (2013).

Es well as defining an endeffector for the new group, specifying the link it is represented by.

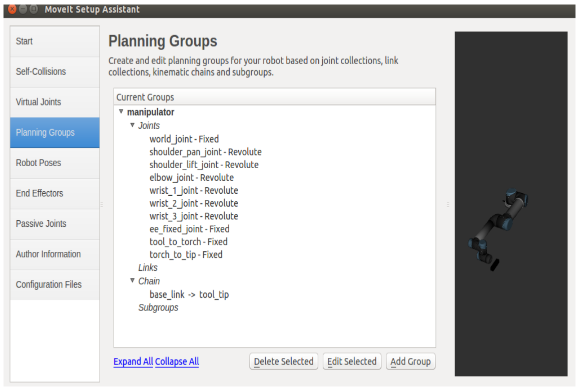
Generating the files will create a moveit config folder containing a number of launch scripts and config files. However this is not enough to simulate the robot.

URDF for joint limited robot

The UR robots have joints capable of turning 2 full rotations. This means they operate within a range of $[-2\pi, 2\pi]$. The move group node, most planning libraries, and algorithms used by ROS are made with industrial robots with joint limits within $[-\pi, \pi]$ in mind.

The URDF file installed with the universal robots package is already coded to set joint limits to $[-\pi, \pi]$ for all joints if the variable joint_ limited is set to true.

Figure 4.8 Define the group you want to configure

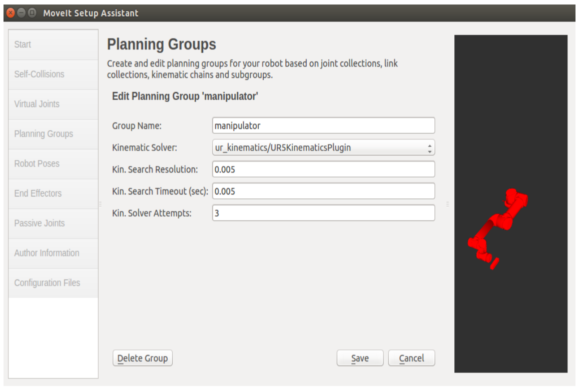


Planning Groups

Create and edit planning groups for your robot based on joint collections, link collections, kinematic chains and subgroups.

```
Current Groups
▾ manipulator
  ▾ Joints
      world_joint - Fixed
      shoulder_pan_joint - Revolute
      shoulder_lift_joint - Revolute
      elbow_joint - Revolute
      wrist_1_joint - Revolute
      wrist_2_joint - Revolute
      wrist_3_joint - Revolute
      ee_fixed_joint - Fixed
      tool_to_torch - Fixed
      torch_to_tip - Fixed
    Links
  ▾ Chain
      base_link -> tool_tip
    Subgroups
```

Figure 4.9 Set the plugin to use as kinematic solver



Planning Groups

Create and edit planning groups for your robot based on joint collections, link collections, kinematic chains and subgroups.

Edit Planning Group 'manipulator'

Group Name:	manipulator
Kinematic Solver:	ur_kinematics/UR5KinematicsPlugin
Kin. Search Resolution:	0.005
Kin. Search Timeout (sec):	0.005
Kin. Solver Attempts:	3

To enable the possibility to change joint limits by launching the move group node and drivers with the ”limited” argument to set joint limits, create an ”urdf.xacro” file identical to the one created earlier except setting the joint_ limited parameter to true after loading the robot.

```
...
<!-- ur5 -->
  <xacro:include filename="\$(find ur5_with_sr300_support)/
     urdf/ur5.urdf.xacro" />

  <!-- arm -->
  <xacro:ur5_robot prefix="" joint_limited="true"/>
  ...
```

Save the file as custom_joint_limited_robot.urdf.xacro. Launch files for both the move group, gazebo and the robot driver can handle the limited argument by loading the limited URDF instead of the original one.

The move group node and the nodes it launches will by default not support the limited argument and need to be altered by hand. Full scripts can be found in appendix section 7.4.5 Using the launch files found in the ur5_ moveit_ config package as examples it should be simple to declare arguments and set up the statement that loads the correct URDF.

Additional configurations and Gazebo

To be able to simulate the robot with the sensor attached it is necessary to make a few scripts providing that functionality. Full scripts are found in the appendix in section 7.4.5.

First, a script for loading the robot description into the parameter server needs to be made.

A script for starting gazebo with the custom robot can be copied from the ur_ gazebo package and slightly altered to upload the custom robot instead of the corresponding UR model.

Create a wrapper for launching the move group node to allow communication interface for joint trajectory execution to be switched between simulation in gazebo and a robot driver with the "sim" argument.

The MoveIt controller manager launch file will not be generated correctly by default. Open the MoveIt config folder generated earlier and modify the file called: robot_ moveit_ controller_ manager.launch.xml to look like shown in the appendix.

4.2.2 Limiting inspection planner workspace

To ensure the path is viable for experiments, the planner must work around obstacles and only generate viewpoints that are reachable by the robot. Meaning they have to avoid singularities, collision and be within the workspace of the robot.

Unobstructed workspace

The ur5 has a reach of approximately 850 mm in a spherical shape around the base. Howver the robot does not operate well if the tcp is within 200 mm of the center of the base. Assuming the sensor extends the tool center point by 45 mm, this range is extended to 250 mm for safety.

For practical purposes the workspace can be considered as a sphere with radius 800 mm not including a cylinder through the base with radius 250 mm with the robot base in the middle, passing through the sphere from bottom to top.

This can be represented by:

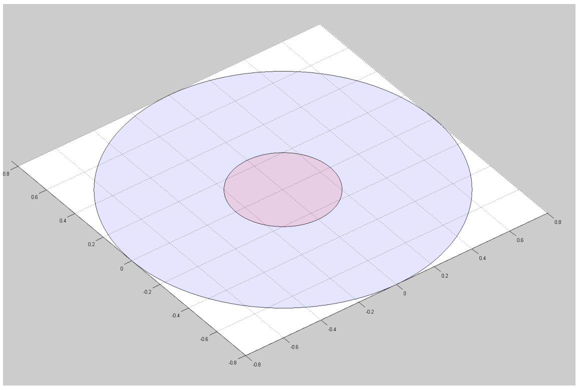
$$x^2 + y^2 + z^2 < (800mm)^2$$

$$x^2 + y^2 > (250mm)^2$$

Due to the nature of the planner this has been simplified and implemented as:

$$X_{MAX} = Y_{MAX} = 0.750m$$

Figure 4.10 Robot workspace



$$X_{MIN} = Y_{MIN} = -0.750m$$

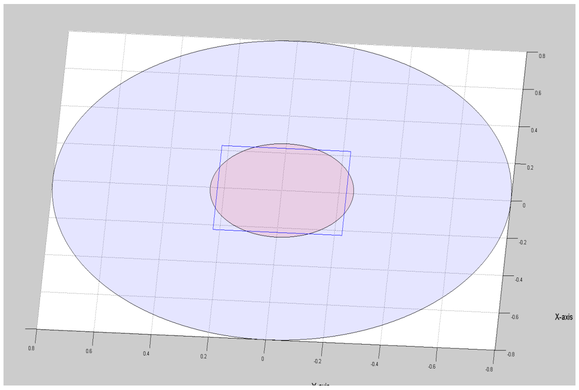
$$Z_{MAX} = 0.750m \quad Z_{MIN} = -0.300m$$

The inner circle is represented as a box obstacle with centre in the origin with dimensions

$$dim_{BOX} = \begin{bmatrix} 0.45 \\ 0.45 \\ 2.0 \end{bmatrix} m$$

The obstacle is shown in figure 4.11 as a blue square. The projected box nearly contains the extended region with minimal unnecessary obstruction.

Figure 4.11 Box obstacle approximating XY constraints projected onto actual constraints

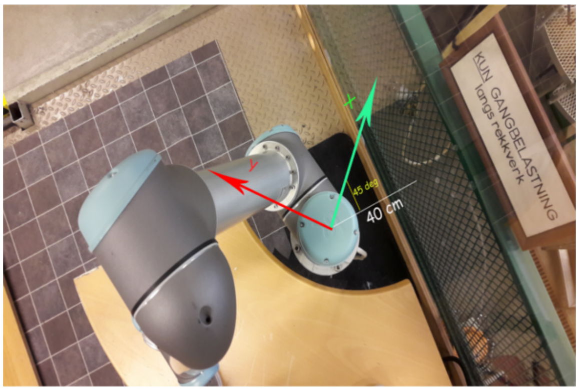


Obstacles

To avoid a path forcing the manipulator to collide with the environment the workspace will be further constrained by adding obstacles and workspace limitations.

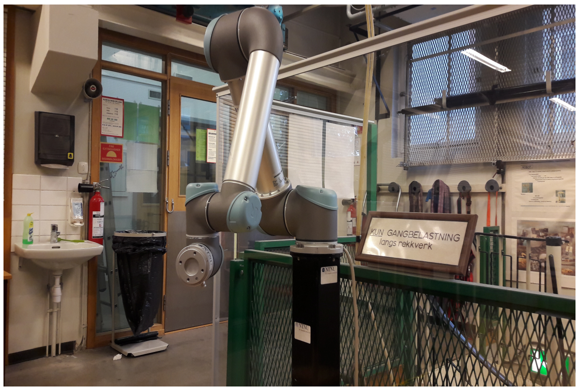
ROS maintains both the environment and the robot model in a planning scene object, which is used to check for collision checks for any given robot state. To make the move group plan while avoiding obstacles they need to be added to this planning scene object.

Figure 4.12 Robot workspace, seen from above. Shortest distance from center of the base to the wall measured to be approximately 40 cm



As seen in the picture there is a table, measured to be at approximately 33 cm below the base of the manipulator. This can be represented either by imposing a workspace limit on the z coordinate of minimum $-30cm$ or by adding a collision object representing the table and its legs. The second approach will restrict the workspace less but results in higher computational complexity.

Figure 4.13 Robot workspace, seen from the side

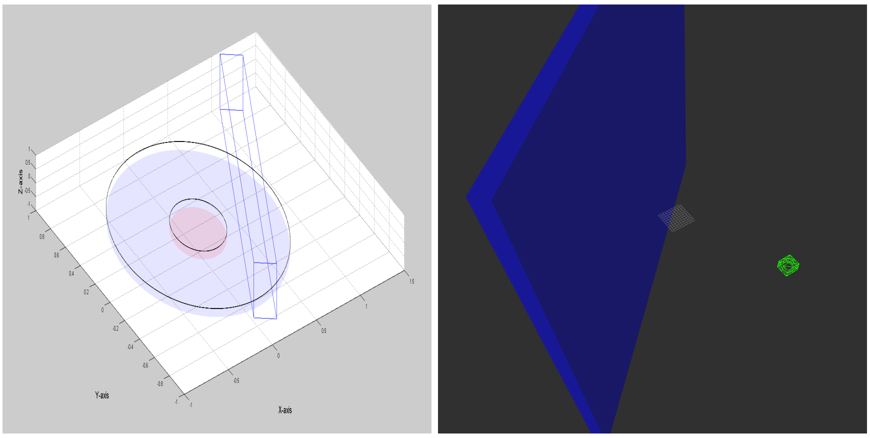


There is also a wall at approximately a 45^o angle, the distance from the base to the wall is approximately $40cm$. Given a 45 degree angle from both x and y axis this means offset for x and y can be described as:

$$x_{wall} = \sqrt{40}cm = y_{wall}$$

Figure 4.14 visualizes the wall obstacle as a blue cube together with the XY limitations of the workspace in Matlab on the left picture. The right picture is of the rviz visualization of the wall. The gray grid placed in the origin of the workspace, and the green cube is placed at coodinates: $[30, -30, 30]$ relative to the robot base.

Figure 4.14 Wall shown in Matlab generated workspace left. Visualized in rviz on the left



Due to the minimal distance criterion and the object being convex, the path should never collide with the object being inspected. However as the inspected object increases in size it may be necessary to include the object in the list of collision object being passed to the planning scene.

Chapter 5

Experiments

To test the system there have been three experiments conducted. First a path optimally, convergence and length comparison between the output plans from the algorithm before and after it is augmented to only include viable poses.

Then experiments simulating a select few paths both with Gazebo and running them on a real UR5 with an Intel Realsense sr300 development kit attached to the end of the wrist of the robot. This is to analyze the accuracy of which the robot reaches the viewpoints and the number of viewpoints reachable with the robot. This is done to identify potential shortcomings of the path generation and robot motion planning.

Lastly, the point cloud data is analyzed. First to check measurement error and consistency. Then two point clouds captured with the sensor attached to the robot are transformed to overlap, the accuracy of point cloud transform and robot transform data is examined. With the intention of model reconstruction the different models are each inspected by atleast one of the generated paths, however model reconstruction was not attempted due to time constraints and error in point cloud transformation.

The simulations were done on a PC running 64-bit Linux Ubuntu 14.04 LTS, with 7,8GiB RAM, and an Intel Core i5-3470 CPU with 3,20GHz x 4.

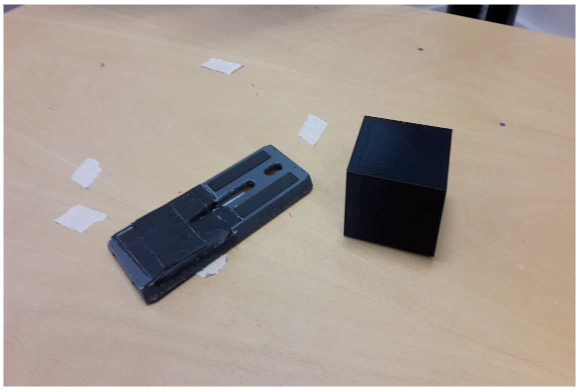
For details on setting up robot simulations and experiments see chapter 4.

Code used to simulate the robot as well as log data can be found at `https://github.com/mmseines/ur5_simulation/tree/master/src`

5.1 Models for inspection and algorithm terminology

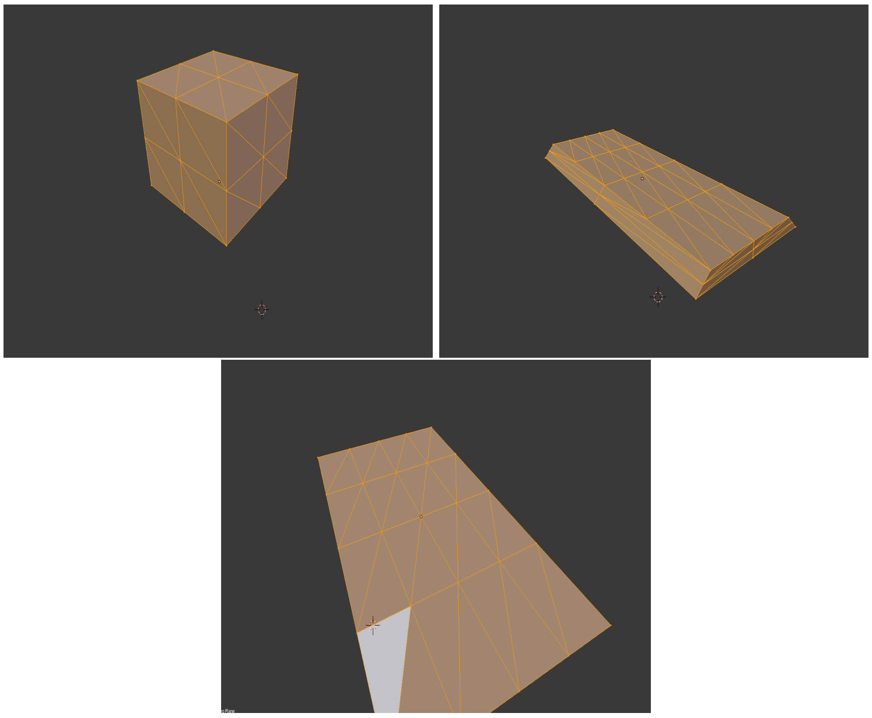
The experiments will be using simple geometric shapes as input meshes. The is done to simplify evaluation of path quality. The box is a 3D printed box with a base of 5.6 cm x 5.4 cm, and height of 4.4 cm. The camera mount is an oblong object with small edges and was selected to be used for specifically for model comparison. The mount has a shape that is easy to model, but it has several artifacts of different types. Such as texture, holes, and errors in curvature.

Figure 5.1 Two simple objects for inspection. 3D printed cube and a old camera mount



For simplifying optimally tests of the algorithm a simple plane of approximately 21 cm x 14.8 cm is used.

Figure 5.2 Models used as input to the planning algorithm



Version	Description
SIP version 0	Besides workspace restrictions and presence of obstacles, this is a 5-dimensional inspection planner without any measures to ensure path viability for a manipulator.
SIP version 1	After sampling viewpoint position, during the grid search to optimize heading, the heading is only considered valid if the resulting 6-dimensional pose has a collision free solution to the inverse kinematics.
SIP version 2	When calculating weights on the edges of the graph formulating the TSP when linear interpolation between the states is performed to check for collision objects, the presence of collision free solutions to inverse kinematics is also checked.
SIP version 3	Combining measures in version 2 and version 3.
SIP version 4	Improvement to version 3 by changing the inverse kinematics check during the grid search to the viewpoint viability criterion described in chapter 3.

Table 5.1: Algorithm versions for testing the effectiveness of algorithm improvements

5.1.1 Planning algorithm variations

To simplify explanation regarding the proposed inspection planning algorithm, its iterations will be divided into several versions. The version are divided like this.

5.2 Algorithm verification

To verify that the resulting paths from the algorithm are both feasible and looks intuitively optimal. The paths generated for inspecting the cube and the plane are compared to intuitive paths observing these objects.

The viewpoint markers visualized in Rviz are the viewpoints sampled during the last iteration of the algorithm. However, the visualized path is always the current optimal path. This means there will frequently be markers that are seemingly not connected to the path whenever the algorithm has multiple iterations without finding a more optimal path. Thus the viewpoint markers visualized are always from the final iteration, and the path might not be.

Planning parameters are selected to match the expected behavior of the sensor and manipulator. Inspection start and end positions for the plane are selected to observe the outskirts of the plane. When inspecting the cube and mount, the start and end positions are selected above the object looking down, to imitate a safe default position, where the sensor can detect the arrival of a new object.

Parameter	Value
Min incidence angle	$\pi/6$
Min distance	35 cm
Max distance	35.1 cm
Translational speed	$15cm/s$
Rotational speed	$1.4rad/s$
Start position[cm]	$[-40, 0, 5]$
End position[cm]	$[-35, 0, 5]$
Model center[cm]	$[-35, 15, -30]$
Horizontal FOV	58 deg
Vertical FOV	45 deg

Table 5.2: Parameters for inspecting the plane with fixed z, pitch and yaw

Parameter	Value
Min incidence angle	$\pi/6$
Min distance	16 cm
Max distance	120 cm
Translational speed	$15cm/s$
Rotational speed	$1.4rad/s$
Start position[cm][rad]	$[-30, 30, 20][0.0, 1.57, 0.0]$
End position[cm][rad]	$[-30, 30, 20][0.0, 1.57, 0.0]$
Model center[cm]	$[-30, 30, -30]$
Horizontal FOV	58 deg
Vertiical FOV	45 deg

Table 5.3: Parameters when planning for cube or camera mount

5.2.1 Optimal path comparison

To investigate the optimality of the inspection planning algorithm, the change in resulting path for inspecting the 3 objects is examined as the allowed number of iterations increase. Estimated path length and trajectory shape are examined and compared to the intuitive solution of the optimization problem. Expected features of an optimal path include little to no backtracking, smooth movement, pushing the limits of observability for parts of the model and that each part of the model is only covered by a subset of points.

The expected path for inspecting the plane would be a sweeping pattern. The shape of the pattern is expected to change dependent on the how much of the plane is covered by the camera from a single position.

The expected optimal motion to inspect the cube is a circular motion around it. Minimizing the radius of the circular motion while maintaining visibility of each side. Each side is only observed once before returning to the default position. The height of the circular motion above the cube is based on the minimal incidence angle parameter and vertical spread of the camera.

The original algorithm by Bircher et al. (2015a) is deterministic and maintains this

Figure 5.3 Expected shape of ideal inspection path of plane

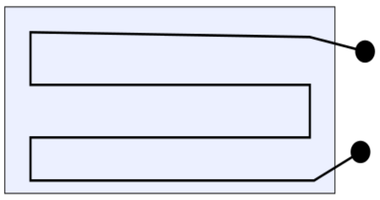
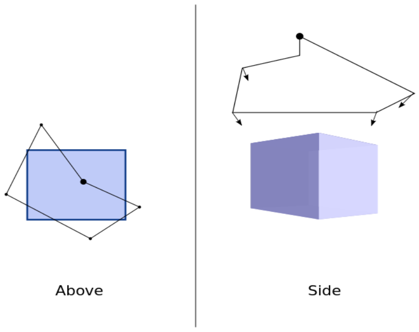


Figure 5.4 Expected shape of ideal inspection path of cube



property after a variable pitch is added. The only variable changing as a result of changing maximum iterations is the smoothing parameter implemented for use with the fixed wing implementation. This was discovered searching the code for an explanation for the results. Code snippet found in section 7.6.2 of the appendix. The path smoothing algorithm shown will replace the normal states used for viewpoint sampling with states that are multiple viewpoints away but still able to observe the same triangle. This causes subgroups of viewpoints to be contracted to form near linear subpaths.

The augmented algorithm solves state validity by finding states is a small neighborhood to the optimal solution, however, as a consequence of random viewpoint sampling, the algorithm is no longer deterministic. Therefore path planning is attempted multiple times for each parameter setting, and both average and the optimal solution is considered for performance evaluation.

To plan for a given model, start by running the planning nod

- roslaunch koptplanner ur5.launch.

This will launch the augmented planner, as well as load the robot description of the robot with the 3D sensor attached as an end effector, this is used in the augmentations to perform collision checks and check the presence of a solution for inverse kinematics.

To visualize the results the ROS gui rviz is run in a separate terminal.

- rosrun rviz rviz.

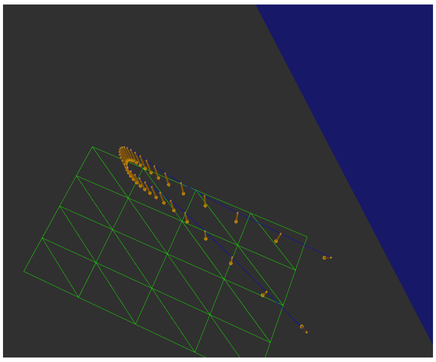
Then to request the planner to make an inspection plan with a given max number of iterations run, this will only load stl files located in the inspection planners designated mesh folder.

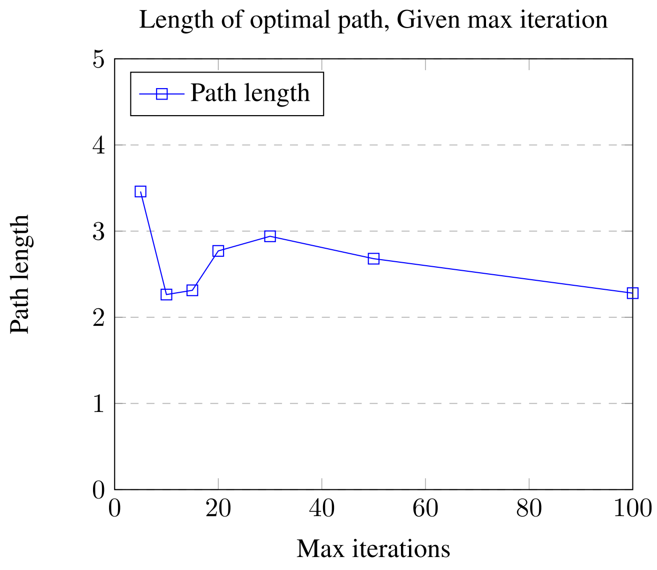
- rosrun requester urRequester _ iter:=NUM _ mesh:=”mesh.stl”

Algorithm v0 optimal inspection path, plane.

The algorithm is first given a two dimensional optimization problem. Both orientation and position on the z axis is artificially locked by constraints.

Figure 5.5 Result after running the algorithm with locked z, pitch and yaw, optimal solution after 100 iterations

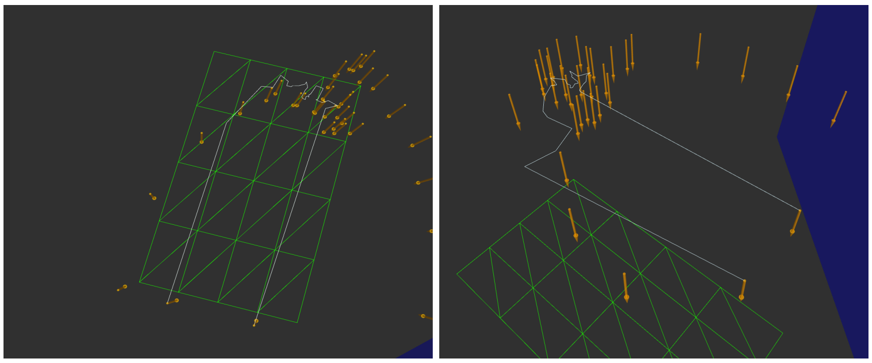


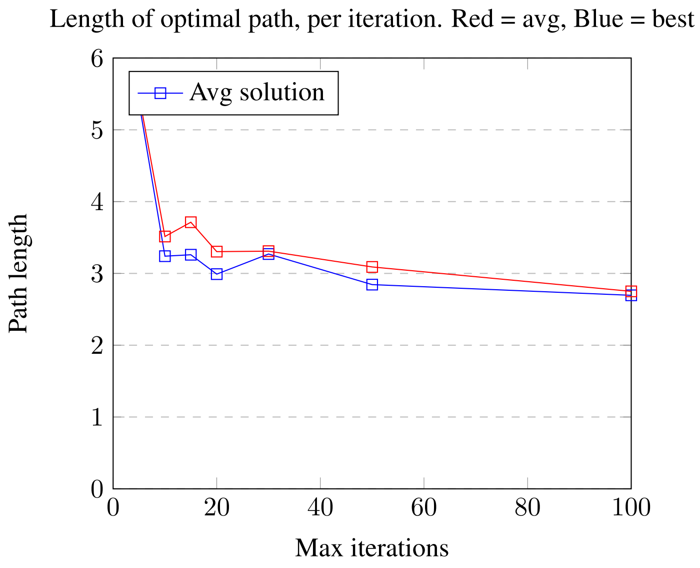


Algorithm v4 optimal inspection path, plane.

Plot containing most optimal path resulting after the given number of iterations. The augmented algorithm appeared to be struggling in the two dimensional space. Generally resulting in paths with most of the viewpoints over the centre of the plane, and paths looking discontinuous between them.

Figure 5.6 Result after running the algorithm augmented and with locked z, pitch and yaw. Two different solutions, solution to the right showcases a more connected solution, where the one to the left only finds valid points in the centre



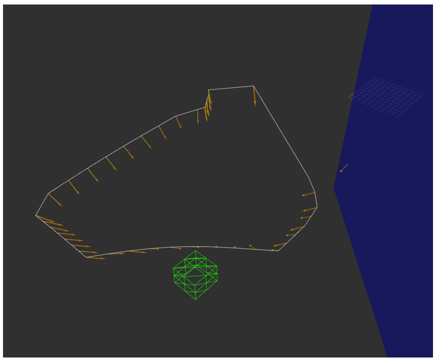

Length of optimal path, per iteration. Red = avg, Blue = best
Avg solution
Path length
Max iterations
0
1
2
3
4
5
6
0
20
40
60
80
100

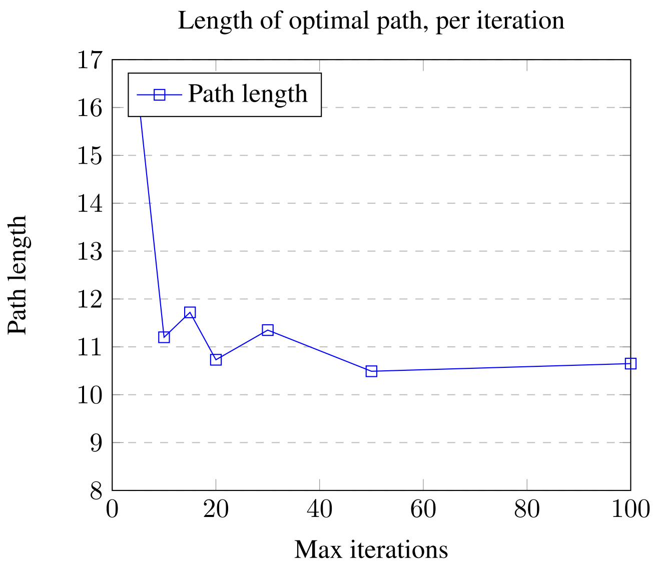
It can be observed that not only is the paths generated significantly longer, they also appear to not have any viewpoints in proximity to the start and end state.

Algorithm v0 optimal inspection path, cube

To expand the problem the optimal path inspection a 3D object with 5 degrees of freedom.

Figure 5.7 Result after running the algorithm as presented in my paper, 50 iterations, optimal solution

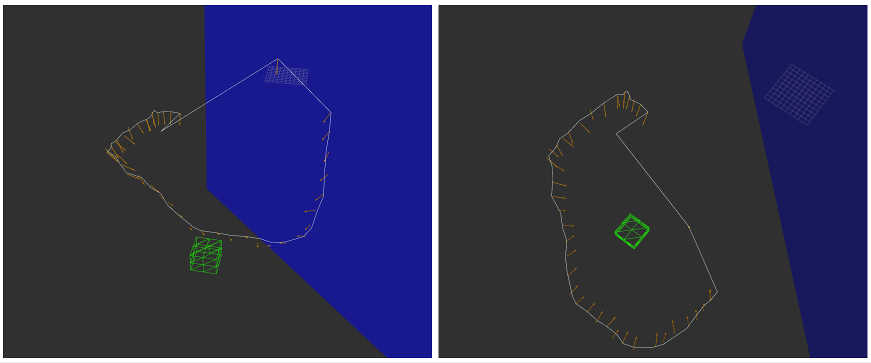


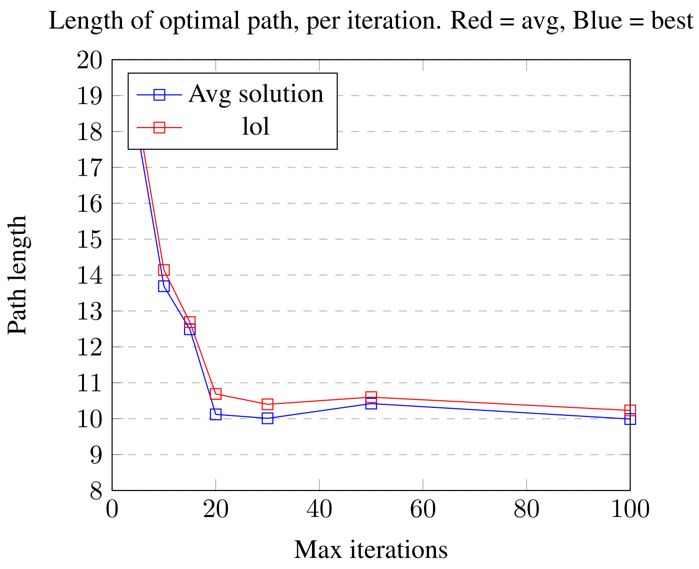


Algorithm v4 optimal inspection path, cube

With 5 degrees of freedom when selecting points, algorithm v4 seems to outperform algorithm v0. Although the path still seems problematic for smooth linear movement, it is estimated to be shorter.

Figure 5.8 Optimal path found with augmented algorithm running 100 iterations, resulting in a path of length 9.99

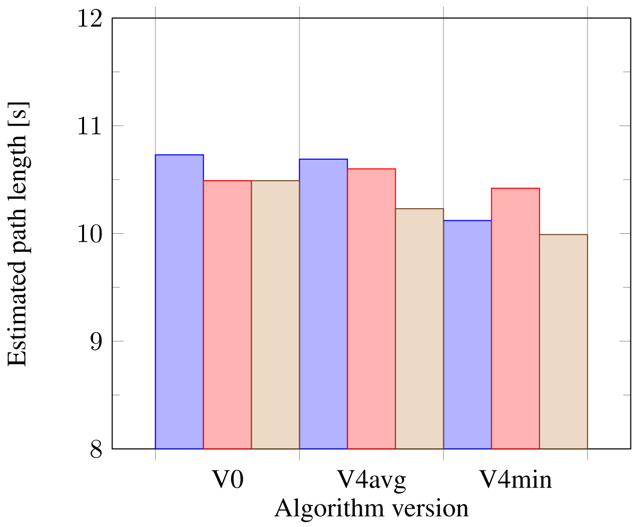




Comparison

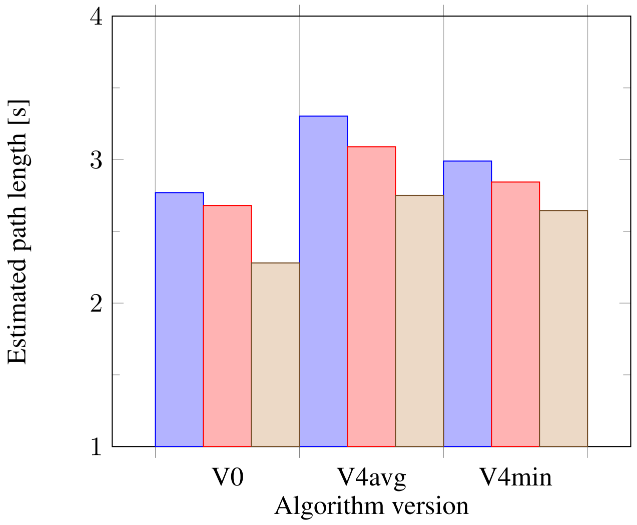
The estimated path length returned by the algorithm variations, V0 refers to the algorithm without any extensions beyond adding variable pitch Seines (2016). V4 refers to the algorithm with all additions covered in chapter 3, why it is referred to as version 4 is revealed later in this chapter. Since algorithm version 4 is not deterministic, both expected value and the optimal value is plotted.

Estimated length of the path inspecting the box.



The average inspection path generated by algorithm v4 is assumed to be shorter than the path generated by algorithm v0 at 20 iterations, longer at 50 iterations and significantly shorter at 100 iterations. The optimal path generated by the augmented algorithm is significantly shorter than the path generated by the unaugmented algorithm for both 20 and 100 max iterations. This implies that the augmented algorithm has high variance but is expected to output paths of similar or less length compared to the first iteration of the algorithm

The estimated length of the path inspecting the plane.



Algorithm v4 returns paths that are longer than those returned by v0. However as seen in the path visualizations, the path generated by the augmented algorithm has no viewpoints in the vicinity of the start or the end position. A possible explanation is that poses in this region do not have valid inverse kinematics solutions.

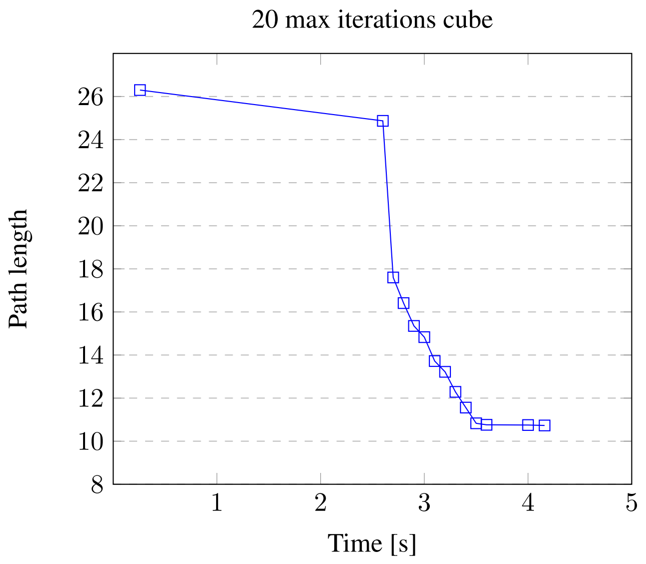
5.2.2 Convergence tests

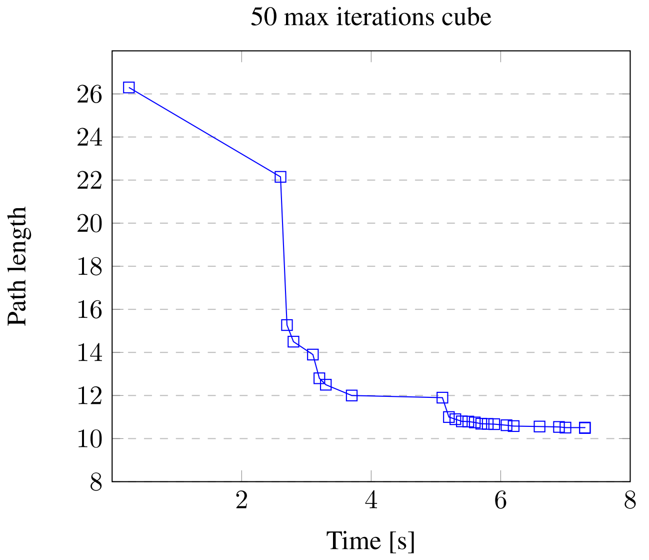
To inspect algorithm convergence, the convergence will be presented for both 20, 50 and 100 iterations when planning inspection for the cube the cube. The purpose of this test is to find out if the algorithm converges towards a given length, and how much the convergence depends on initial states and random samples if it does.

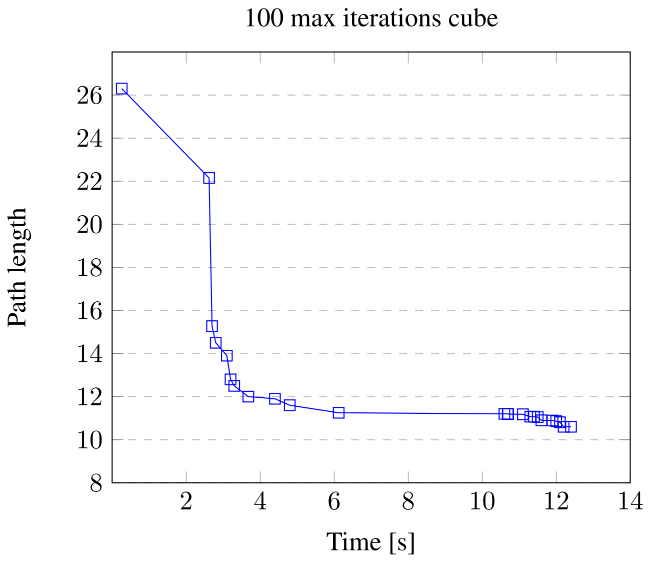
For this test only paths generated when planning in 5 dimensions for the cube will be considered as the algorithm is intended to plan motions in 5 dimensions.

Algorithm v0 convergence test

As the original algorithm is deterministic, each variation was run only once and plotted below.

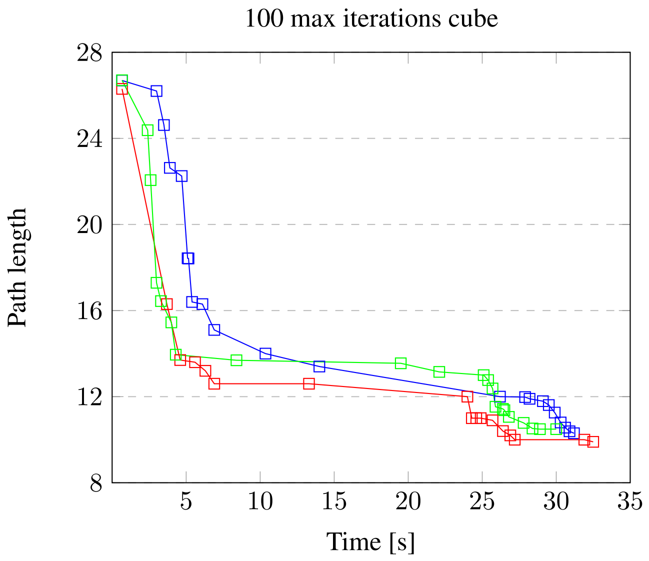


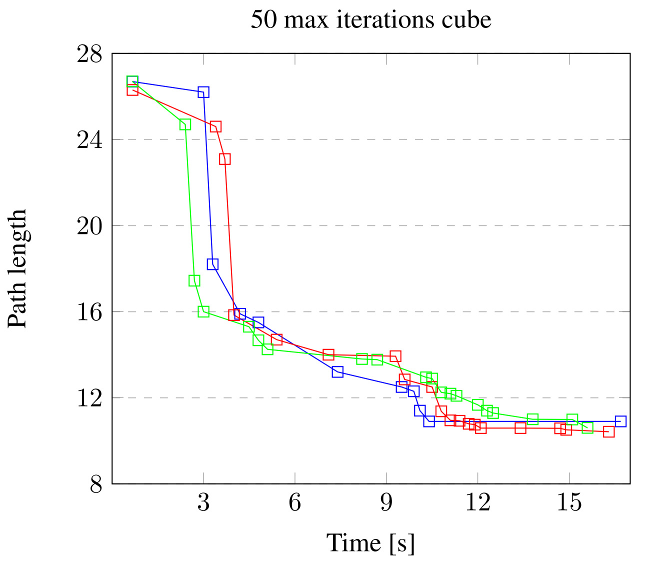


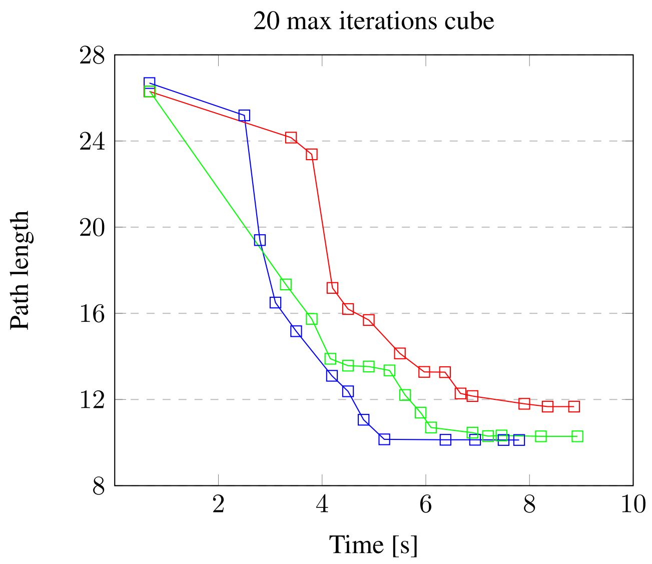


Algorithm v4 convergence test

The augmented algorithm would frequently give different solutions to a given path with the same parameters. To get information on how the viability criterion affects algorithm convergence, the algorithm is run 3 times for a given set of parameters. The convergence of the inspection plan resulting from each run is plotted together.


100 max iterations cube
Path length
Time [s]
28
24
20
16
12
8
5
10
15
20
25
30
35


50 max iterations cube
Path length
Time [s]
28
24
20
16
12
8
3
6
9
12
15



Comments

In all cases both algorithms have similar responses during the first few seconds, The path starts with no improvement the first few seconds, then rapidly converge towards what appears to be some local minimum. Followed by asymptotic improvement the remaining iterations.

5.2.3 Complexity

To get an idea of algorithm complexity, the same model with different resolutions will be inspected and the time spent running the algorithm will be examined.

three cubes with different resolution and identical size is run with 20 max iterations.

Of note, the time evaluation disregards time spent writing the results to file and the time spent publishing the mesh and path, which are both considerably long, especially as the number of triangles in the mesh increases.

It is worth mentioning that this will depend on available CPU power, to mitigate the random effect CPU activity had on this measure the numbers used are averages.

increasing resolution of the mesh

To increase the resolution of a mesh, open the mesh in a 3D editing software such as blender. Selecting the entire object, there should be an extrusion option called subdivide, shown in the picture. This will divide each triangle into 4 smaller triangles.

As the original cube had a resolution of 40 triangles, running subdivide we get a cube with a resolution of 160 and 640 triangles.

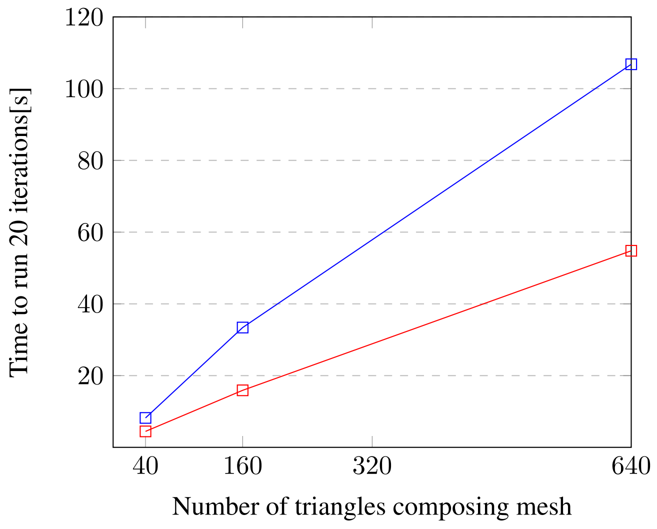
Version	40 triangles	160 triangles	640 triangles
v0	4 450 ms	15 943 ms	54 879 ms
V4	8 256 ms	33 402 ms	106 860 ms

Table 5.4: Computation time for same mesh with varying resolution

Results

The bare bones algorithm and the augmented algorithm are run with the same cube with increasing resolution. Each resolution is ran three times on each algorithm to minimize the effect of background processes and CPU load on the total time estimate.

Total time as a function of the number of triangles composing the mesh



The actual time is given in the table below.

The blue plot is the time spent for the unaugmented algorithm, and the red plot is the time spent for the augmented algorithm. Time consumption appears to increase linearly with the resolution of the mesh for both versions of the planning algorithm. The augmented algorithm appears to spend twice the amount of time of the unaugmented algorithm to generate a path.

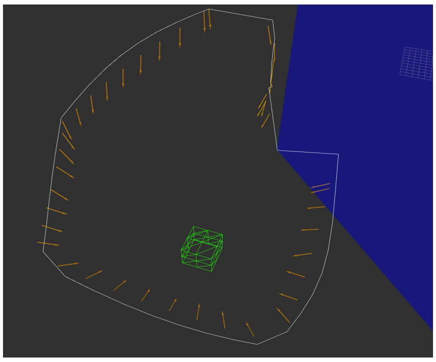
The number of distinct viewpoints in the resulting path seems to always be equal to the number of triangles forming the mesh.

5.2.4 Investigating varying convergence

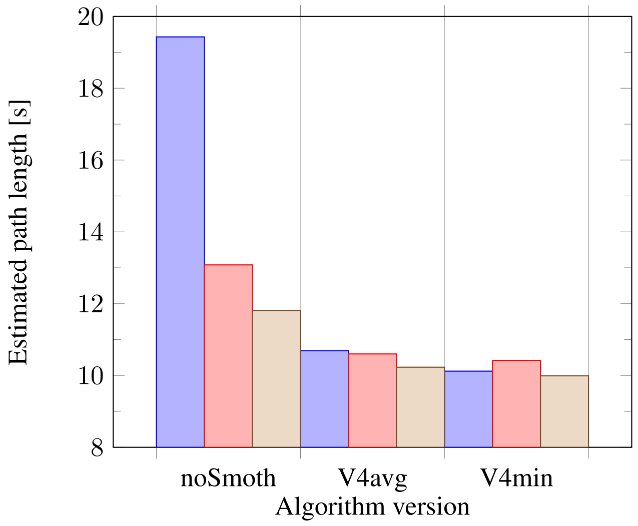
The experiments for path length and convergence showed that algorithm behaviour changed drastically based on the number of maximum iterations, even though it is not supposed to.

After the experiments, the source for different algorithm behavior based on maximum iterations was discovered. Even though the smoothing algorithm for fixed wing AUVs is not supposed to compile, it has been during the experiments. The smoothing algorithm is intended to shape the path to fit the maneuverability of a fixed wing AUV, however, it appears to help minimize path length. The algorithm returned consistently longer paths when the smoothing algorithm was removed.

Figure 5.9 Path generated with 100 iterations removing the smoothing parameter



Path length of the path resulting from 20 iterations, 50 iterations and 100 iterations. The algorithm appears to converge a lot slower without reaching the same path lengths reached when the path is smoothed. For this reason the path smoothing criterion was considered beneficial to algorithm performance, and the experiments where not repeated without it.



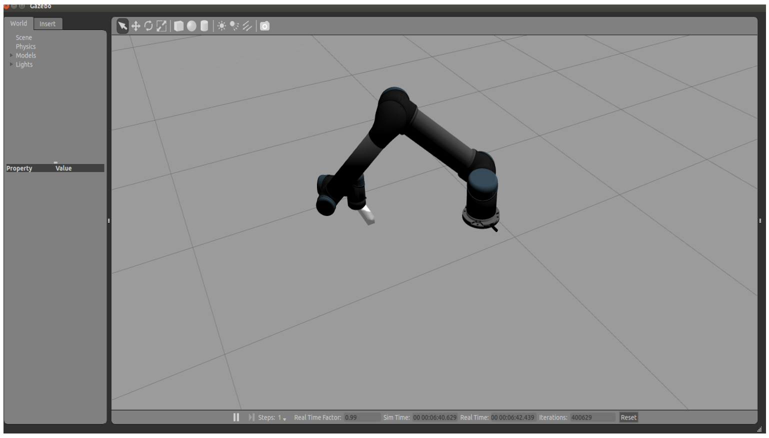
5.3 Simulation and experiments

To test the viability of the plans they are simulated in gazebo. To identify motion planning approach yielding the best manipulator behavior the most promising paths are simulated with either the Cartesian motion planning approach or the Inverse Kinematics Motion Planning(IKMP) approach, both described in section 3.2.2.

5.3.1 Robot simulation

For simulation we use Gazebo, see picture.

Figure 5.10 Robot in the default initial state specified above



To launch Gazebo, the limited argument is added because MoveIt has unpredictable behaviour with joints outside $[-\pi, \pi]$, but it is not necessary.

- roslaunch ur5_ with_ sr300_ gazebo ur5.launch (limited:=true)

This will launch an empty world containing a UR5 manipulator with a square box representing the SR300 sensor attached to the end effector. This is necessary to provide both an accurate visual representation of sensor position as well as loading collision data for detecting possible collisions between the sensor and robot joints.

To launch the robot control interface start the Move Group node. The sim argument specifies that the node is controlling a simulation in Gazebo and not an actual robot. If Gazebo is launched with the limited robot, this should be as well.

- roslaunch ur5_ with_ sr300_ moveit_ config ur5_with_sr300_ planning_ execution.launch sim:=true (limited:=true)

Then one of the motion planning services described below can be launched.

5.3.2 Cartesian motion planning service

To analyze the feasibility of the Cartesian motion planning service, the resulting paths from the augmented inspection planning algorithm is executed using the Cartesian motion planning approach from subsection ”Motion planning in tool space” in section 3.2.2.

For full implementation see the planner.cpp file `https://github.com/mmseines/ur5_simulation/tree/master/src/plan_pkg/src`.

Object	Avg error	Max error	Min error	Total time
Mount	0.1135 mm	0.32 mm	0.027 mm	32 245 ms
Cube	—	—	—	—
Plane	—	—	—	—

Table 5.5: Results simulating with the Cartesian motion planner

The purpose of this test is to see if the given path along with the motion planner provides a safe trajectory for the robot, allowing the sensor to hit all waypoints with the maximal accuracy.

To run the code follow the steps to launch Gazebo and the robot control interface (Move Group node), open another terminal and type the command below to launch the motion planner.

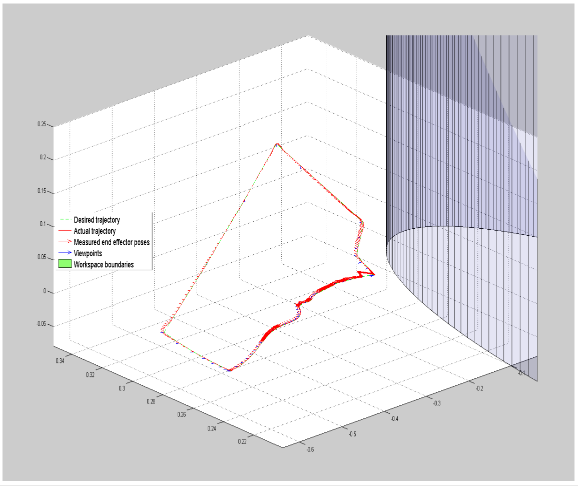
- rosrun plan_ pkg sim _ eef_step:=0.01 _ jt:=0.0 path:=”pathname.csv”

The *eef_ step* argument sets the step length used when interpolating end effector position in cartesian space. *jt* is short for jump threshold but it is set to 0.0 to disable it by default.

Results

Making this algorithm output a viable path is challenging. The path quality depended significantly on start state as well as parameter values. Out of the three paths, only the inspection plan for the mount executed without fail.

Figure 5.11 Results for the mount inspection plan eef_ step = 0.03 and jump treshold = 0.0



5.3.3 Inverse kinematics Motion Planner

The inverse kinematics point to point based motion service was implemented to ensure robot configurations where chosen to be sensible with regards to minimizing robot motion. As well as robustness in completing a path by handling singularities and possible collisions.

For full implementation see the ikplanner.cpp file `https://github.com/mmseines/ur5_simulation/tree/master/src/plan_pkg/src`.

To launch the service, make sure Gazebo and the Move Group node are already running in separate terminals, then open a terminal and launch the motion planner with the following command.

- rosrun plan_ pkg ikSim _ path:=”path.csv”

To analyze the feasibility of the inverse kinematics based motion planning service the optimal path for the cube, the camera mount and plane are simulated in Gazebo, and the joint trajectory is logged. This is used to perform forward kinematics giving both end effector position and orientation.

The purpose of this test is to compare both accuracy and speed to the Cartesian motion planner as well as to determine if the planner and the paths make a suitable combination to control a real robot.

Results

Figure 5.12 Results when simulating augmented algorithm generated path for the plane

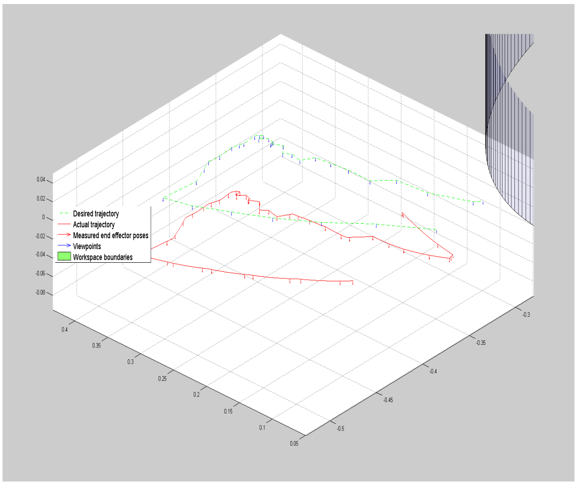


Figure 5.13 Results when simulating augmented algorithm generated path for the mount

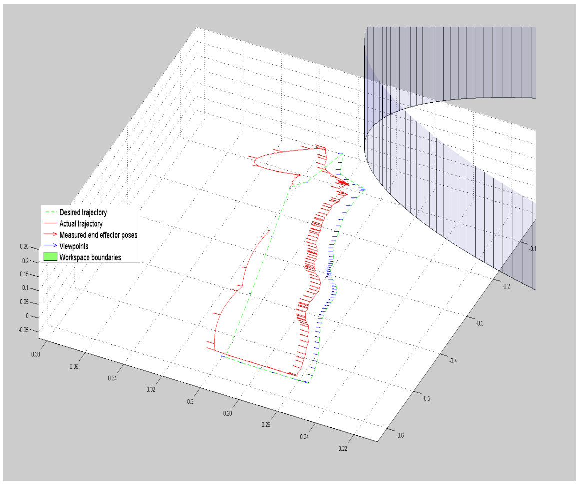
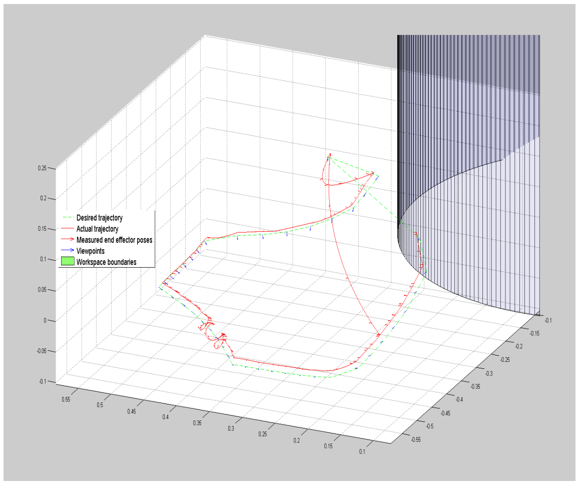


Figure 5.14 Results when simulating augmented algorithm generated path for the cube



Object	Avg error	Max error	Min error	Total time
mount	9.0 mm	10.5 mm	0.6 mm	43 135 ms
cube	9.1 mm	10.4 mm	2.8 mm	53 333 ms
plane	47.6 mm	24.4 mm	51.7 mm	23 399 ms

Table 5.6: Results when simulating with IKMP inspecting all objects

While the inverse kinematics based motion planner has significantly lower accuracy and is slower, it manages to simulate a collision free path for all three objects.

Inspection planning variations

To estimate the effect of the measures taken to ensure the inspection planner provides manipulator friendly paths, inspection planning algorithm is divided into multiple versions. Each version is used to generate a plan with the same parameters and input, such that the only difference is what measures implemented in that algorithm version.

Algorithm versions are iteratively improved as described in section 5.1.1.

Figure 5.15 Results when simulating path resulting from inspection planning algorithm v0

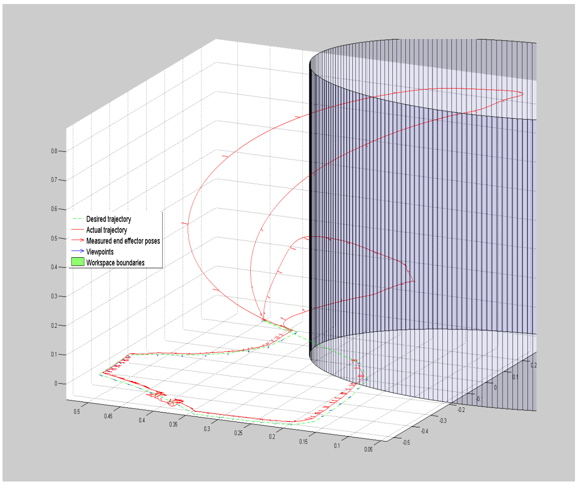


Figure 5.16 Results when simulating path from inspection planning algorithm v1

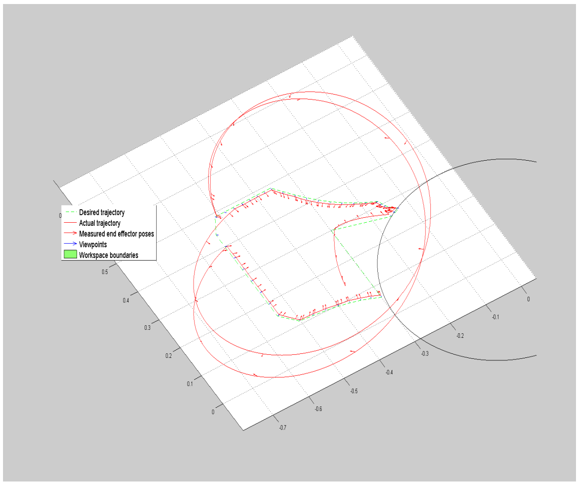


Figure 5.17 Results when simulating path resulting from inspection planning algorithm v2

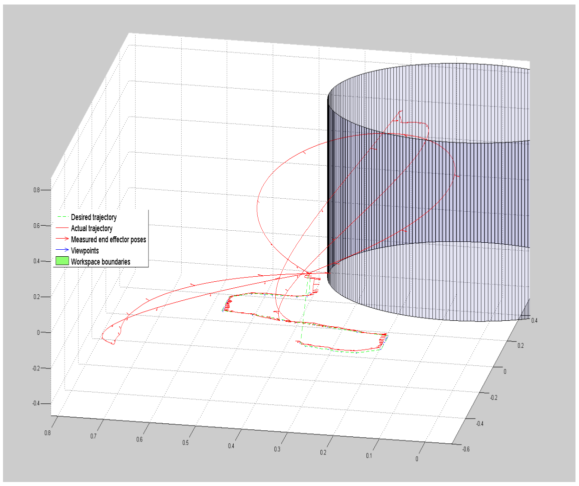


Figure 5.18 Results when simulating path resulting from inspection planning algorithm v3

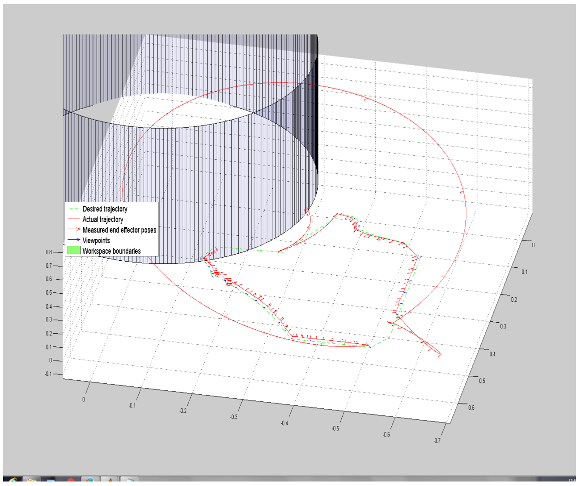
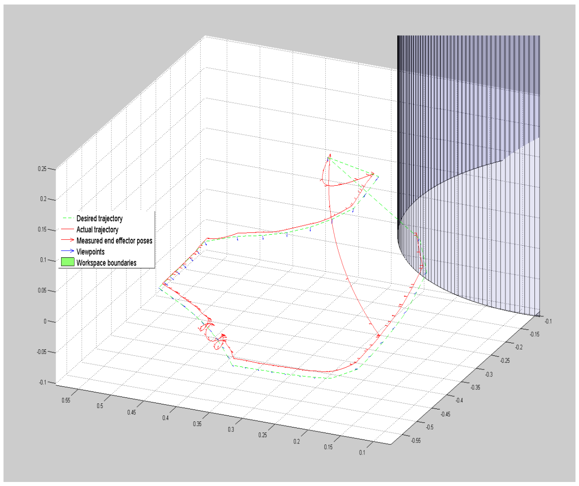


Figure 5.19 Results when simulating path resulting from inspection planning algorithm v4



To compare the results beyond the position of the sensor over time. The number of viewpoints that were considered to be reached by the motion planner, total time from start to end and the average error in millimeter from the path is given in the table below.

Version	Completion	Total time	Avg error	Total path length
v0	41/41	87 452 ms	9.0 mm	4.6227 m
v1	39/41	45 323 ms	10.7mm	7.719 m
v2	41/41	67 124 ms	8.3 mm	7.1049
v3	40/41	47 976 ms	9.78 mm	4.89 m
v4	41/41	53 333 ms	9.1 mm	1.7633 m

Table 5.7: Results in path completion, total time executing and total lenght

The algorithm without any improvements performed better with regards to minimizing actual distance traveled than any other version except version 4. However all improvements reduced the total inspection time significantly. The simple inverse kinematics check appears to not work as intended as the only paths where one or more viewpoints could not be reached where paths generated by inspection planning algorithms with the check. Long arcs between viewpoints are present in the trajectories from every inspection plan except those generated by the fourth version of the inspection planning algorithm. The trajectories for inspection of the camera mount 5.12 and the plane 5.13 are free of long arcs as well.

Algorithm comparisons

To compare quality of motion planning service algorithms, they are plotted together and errors are measured up against one another for the augmented mount path.

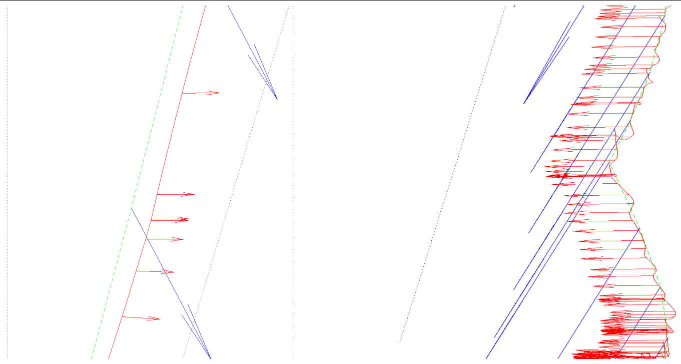
As far as performance goes the Cartesian motion planning service is far superior in both accuracy and total time of execution. However, the inspection planning algorithm does not return paths capable of generating a robot trajectory with the Cartesian motion planning service.

Neither of the algorithms achieves the expected orientation at each viewpoint according to the robot state logger, as seen in the close-ups of the path pictures below. However, this is assuming there is no error with the end effector position and orientation measurements.

Algorithm	Avg. error	Max error	Min error	Total time
IKMP	9.0 mm	10.5 mm	0.6 mm	43 135 ms
Cart	0.1135 mm	0.32 mm	0.027 mm	32 245 ms

Table 5.8: Comparison between Cartesian motion planner and IKMP performance simulating inspection plan for the camera mount

Figure 5.20 IKMP and the left, Cartesian motion planner on the right



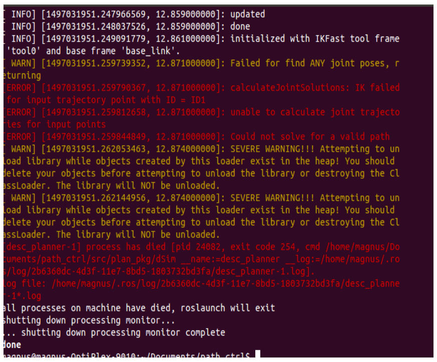
5.3.4 Descartes, dense planning algorithm

Proposed as a more robust, and optimal version of the Cartesian path planner, the Descartes package implements an algorithm that seems to be ideal for this type of application in the dense planner.

However, the Descartes package proved to be problematic to integrate with the ur5. In the end, time restrictions made it unfeasible to spend a large amount of time trying to figure out where the mistake in the implementation was.

Below you can see the resulting output when trying to generate a trajectory with the Dense planner.

Figure 5.21 Result when running the dense planner launch files.



```
[ INFO] [1497031951.247966569, 12.859000000]: updated
[ INFO] [1497031951.248037526, 12.859000000]: done
[ INFO] [1497031951.249091779, 12.861000000]: initialized with IKFast tool frame
 'tool0' and base frame 'base_link'.
[ WARN] [1497031951.259739352, 12.871000000]: Failed for find ANY joint poses, r
eturning
[ERROR] [1497031951.259790367, 12.871000000]: calculateJointSolutions: IK failed
 for input trajectory point with ID = ID1
[ERROR] [1497031951.259812658, 12.871000000]: unable to calculate joint trajecto
ries for input points
[ERROR] [1497031951.259844849, 12.871000000]: Could not solve for a valid path
[ WARN] [1497031951.262053463, 12.874000000]: SEVERE WARNING!!! Attempting to un
load library while objects created by this loader exist in the heap! You should
delete your objects before attempting to unload the library or destroying the Cl
assLoader. The library will NOT be unloaded.
[ WARN] [1497031951.262144956, 12.874000000]: SEVERE WARNING!!! Attempting to un
load library while objects created by this loader exist in the heap! You should
delete your objects before attempting to unload the library or destroying the Cl
assLoader. The library will NOT be unloaded.
[desc_planner-1] process has died [pid 24082, exit code 254, cmd /home/magnus/Do
cuments/path_ctrl/src/plan_pkg/dSim __name:=desc_planner __log:=/home/magnus/.ro
s/log/2b6360dc-4d3f-11e7-8bd5-1803732bd3fa/desc_planner-1.log].
log file: /home/magnus/.ros/log/2b6360dc-4d3f-11e7-8bd5-1803732bd3fa/desc_planne
r-1*.log
all processes on machine have died, roslaunch will exit
shutting down processing monitor...
... shutting down processing monitor complete
done
magnus@magnus-OptiPlex-9010:~/Documents/path_ctrl$
```

Full code and how it was installed is included in the appendix.

5.3.5 Motion planner experiments with UR5

To test if the simulations are accurate and if the paths are executable on a real manipulator without triggering safety locks and similar features.

The accuracy of the robot motion compared to the simulations is tested by having two paths with safe behavior for the cartesian motion planner and the inverse kinematics point to point planner run on a robot and compared to the response on the simulated trajectories.

As the robot and control box is not compatible with the standard ur_ driver package, a modern driver by Andersen (2015) will be used instead. The source is downloaded from `https://github.com/ThomasTimm/ur_modern_driver` and compuled using catkin.

The default driver configuration does not support blocking path execution calls, meaning it will fail when attempting to run multiple joint trajectories sequentially. This is a problem for the IKMP as the program relies on the move group node for checking completion of each path.

- roslaunch ur_ modern_ driver ur5_ bringup.launch robot_ ip:=255.255.255.255

The compatible configuration however, will work as it supports blocking trajectory execution calls.

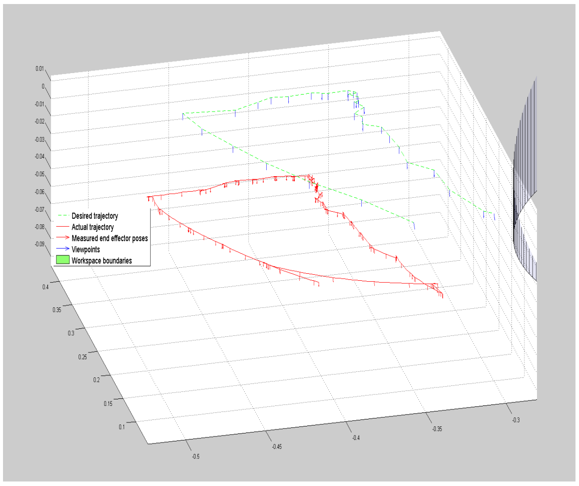
- roslaunch ur_ modern_ driver ur5_ bringup_ compatible.launch robot_ ip:=255.255.255.255

Inverse kinematics planner with compatible driver

Since the point to point approach used by the inverse kinematics motion planner is only compatible with the compatible driver.

The inspection path for the plane tested with the inverse kinematics motion planner, is run on the robot yielding the path below.

Figure 5.22 Measured trajectory, running the inverse kinematics based planner



While it may not be apparent in the plot from the side, the trajectory as seen from the top reveals the overshoot resulting from the jittery point to point movement.

While the path is similar to the simulated path, a significant difference is a frequent overshoot when approaching each viewpoint. While the difference in path length can be a

Figure 5.23 Picture from above, showing overshoot from motion control. Overshoot looks like small branches from the main path

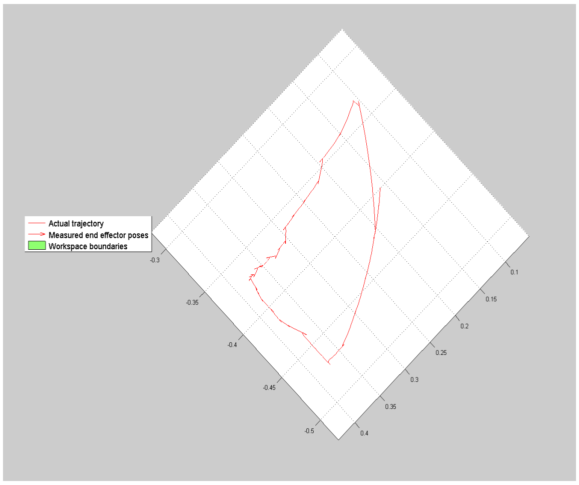
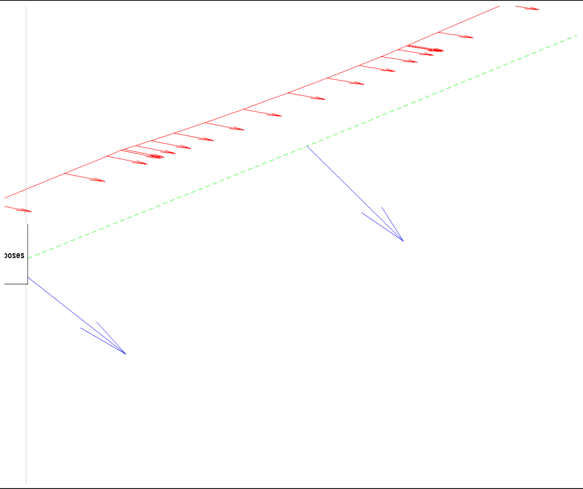


Figure 5.24 Error in achieving desired orientation. blue arrows are the specified orientation, measured orientation given by the red arrows



symptom of this, but the manipulator had unplanned movement towards the start position before reaching the end state.

Cartesian planner with compatible driver

As the cartesian motion planner produces a single trajectory, it is compatible to use with either of the drivers. However to keep the results consistent the compatible driver was chosen for the simulations.

Enviroment	Avg error	Min error	Max error	Total time	Total length
Online	49.0 mm	39.8 mm	52.3 mm	24 899 ms	0.9743 m
Simulation	47.6 mm	24.4 mm	51.7 mm	23 399 ms	0.7156 m

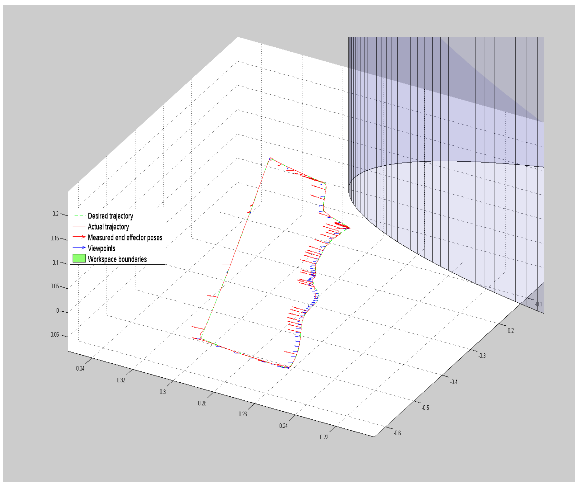
Table 5.9: Comparison between online and simulation performance

Enviroment	Avg error	Max error	Min error	Total time	Total length
Online	3.1 mm	17.5mm	0.0623 mm	32 545 ms	1.2914 m
Simulation	0.1135 mm	0.32 mm	0.027 mm	32 245 ms	1.2828 m

Table 5.10: Comparison between online and simulation performance

Video can be found in attachments in the video folder.

Figure 5.25 Measured trajectory, camera mount inspection with cartesian motion planner



While position is very accurate there seems to be a problem with achieving correct rotation. By zooming in on the path error in rotation is clearly visible and measurement error or possible controller step is visible.

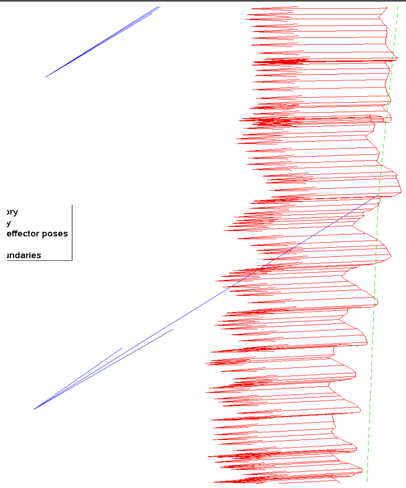
The cartesian planner misses the viewpoints by a larger margin when its controlling a real robot compared to the simulation. It is still showing a much higher accuracy than the inverse kinematics based planner.

5.4 Point cloud reconstruction

To evaluate the possibility of using the sr300 for quality control, the inspection should be accurate enough to be able to remodel the object with some degree of accuracy.

In this experiment, a single point cloud output from the sensor, where the majority of the cube is visible, is first compared to the cubes measurements. This is used to estimate

Figure 5.26 Visible fluctuations from the path as well as orientation error. Blue arrows are the specified orientation, measured orientation given by the red arrows



the accuracy of the sensor. The point clouds captured are transformed to a fixed world frame, using the inverse of the transformation from the world frame to the tool tip link.

The sensor driver is run together with other functionality such as computing surface normals and adding color to the point cloud. To start the node, open a terminal and run the command.

- roslaunch realsense_ camera sr300_ nodelet_ rgbd.launch

This will start the camera and publish point clouds without color to the topic $/camera/depth/points$ and point clouds that have been added color data to $/camera/depth\_registered/points$.

Logging the data is done by opening a separate terminal and running

- rosrun point_ cloud_ io write _ topic:="topic of choice" _ folder_ path:="folder of choice"

The topic argument specifies the point cloud topic to log, and the folder path argument specifies where to save the logged point clouds.

5.4.1 Intel Realsense SR300 sensory tests

To get a baseline of expected sensory input, a still image with the sensor facing one of the corners of the cube is used as a baseline. Sensor input is visualized by a screenshot of an example program for visualizing colored point clouds in realtime.

A point cloud from the same angle is shown in blender below, after being logged and written to a ply file, after filtering nondetermined and infinite values.

As observed the sensor is able to get information of the top of the cube, but the depth information on the sides are sparse. A series of faulty measurements can be seen above the cube as if there is something placed atop of it.

Figure 5.27 Screen shot of realsense point cloud visualization example code, observing the cube

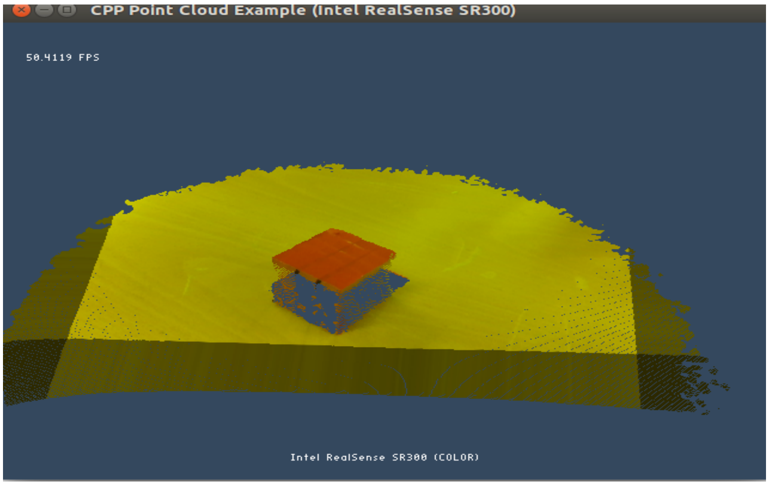
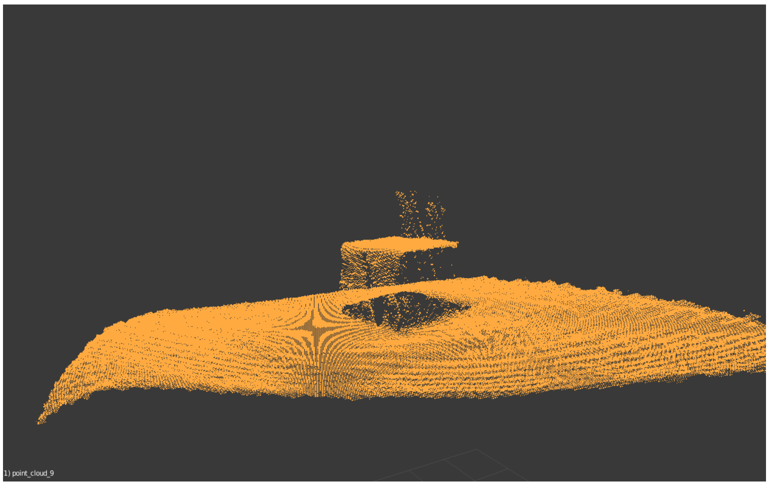

CPP Point Cloud Example (Intel RealSense SR300)
50.4119 FPS
Intel RealSense SR300 (COLOR)

Figure 5.28 Point cloud after transformation, written to stanford(.ply) file, rotated manually in blender



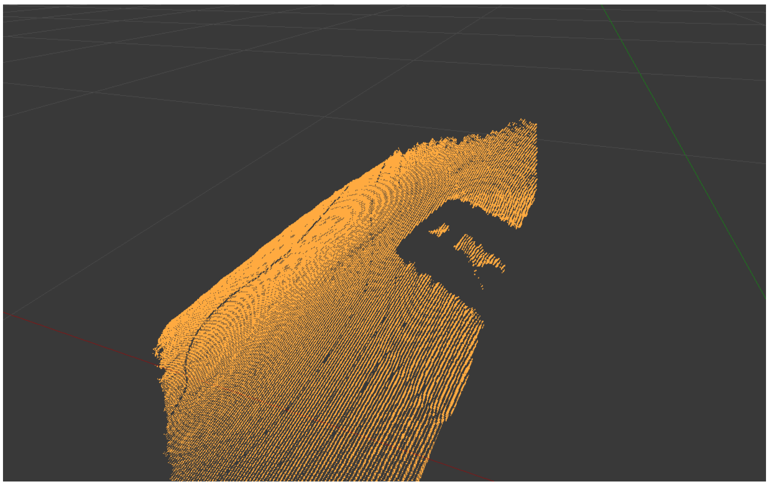
The sensor was also largely unable to observe the mount.

The gaps along the top of the camera mount are visible while the dark plastic around is not.

5.4.2 Point cloud transformation tests

To simplify point cloud reconstruction, it would be ideal to transform each captured cloud into a fixed reference frame, ideally in such a way that the point clouds would overlay one

Figure 5.29 Point cloud captured during execution of cartesian trajectory, to the right: missing data in the shape of the camera mount



another.

The transformation is done before the point cloud is logged to file, by transforming it with the transformation from the sensor frame to the fixed world frame. Given as the inverse to the global link transform to the sensor tip.

However viewing the point clouds in Blender, point clouds do not align. Especially the table does not have the same orientation or position from frame to frame, it is therefore likely that the attempted implementation of point cloud rotation has not been done successfully, or the mismatch between successive frames is due to an error in the state-logger.

Single point cloud.

The sensor was attached to the robot, where the robot performed simple linear motions programmed using the teach pendant.

As visible in this picture, the point cloud transformation does not succeed in transforming the clouds to a fixed frame.

When testing a path involving non constant orientation, successive frames did not match orientation or offset.

Figure 5.30 Point cloud after transformation, written to stanford(.ply) file, pictured in Blender

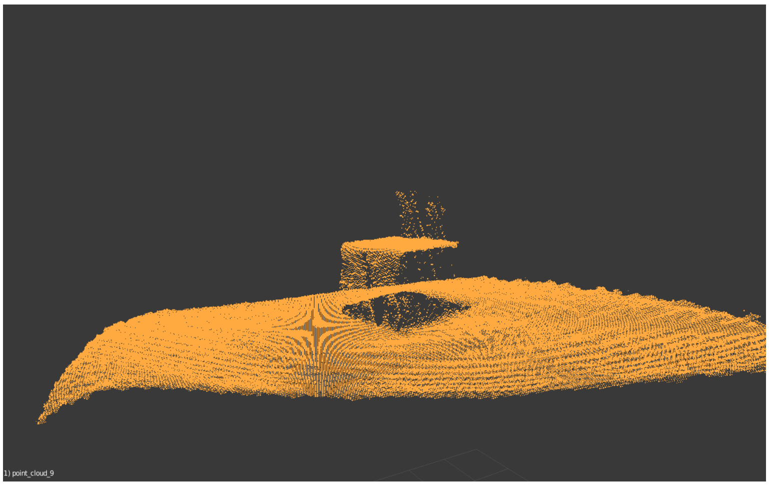


Figure 5.31 Simple linear trajectory to test performance of pointcloud transformation

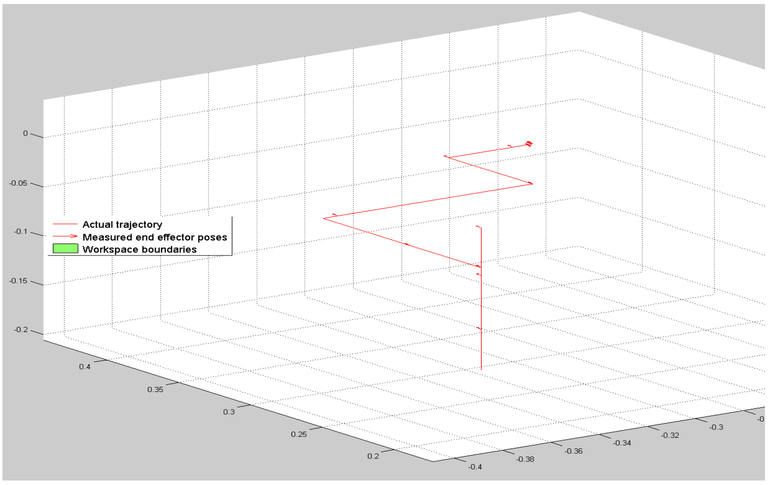


Figure 5.32 Same point cloud together with frame captured 1.3s later from a different angle, pictured in Blender

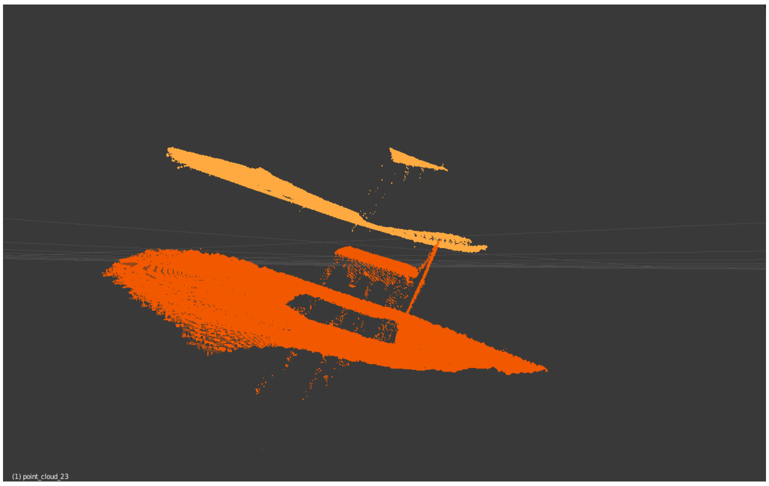
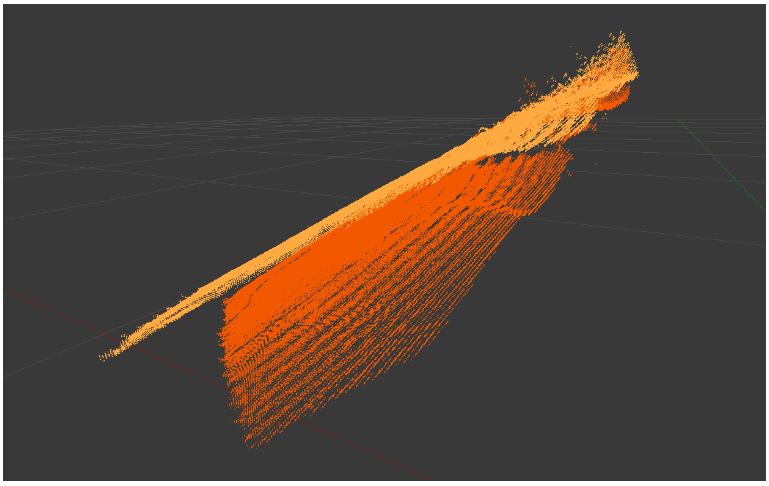


Figure 5.33 Same point cloud together with frame captured 0.7s later from a different angle, pictured in Blender



Chapter 6

Analysis

6.1 Evaluation of inspection planning algorithm improvements

The main goal of this work was improving the inspection planner proposed in Seines (2016) such that resulting paths were well suited for manipulator movement. As well as implementing a motion planning approach accurately executing the inspection paths in order to accurately scan an object, and reconstruct a 3D model from the measurement data.

6.1.1 Path quality

The augmented algorithm adds random sampling to a previously deterministic algorithm, making resulting path length depend on the random states being sampled during execution. The distance evaluation function used when solving the TSP is updated to penalize manipulator collision or lack of feasible solutions from inverse kinematics along the path. As the update to the distance estimation function only includes implementing penalties, lower estimated path length will mean a shorter path.

As the augmentations constrain the search space by demanding that each viewpoint should have a feasible solution to the inverse kinematics. As such it is natural to assume that introducing these augmentations would cause increased path length as opposed to reduce it.

However, the augmented inspection planning algorithm produced paths 5.8 shorter than those of the unaugmented algorithm 5.7 for inspecting the cube. It appears the viewpoint feasibility criterion improves optimality of the inspection algorithm, especially as the number of max iterations increases (above 50). This could mean the deterministic algorithm is finding locally optimal solutions, not global solutions. Another explanation is that the random sampling and motion penalties force changes to the optimal path, thus increasing the amount of the search space being explored.

The same can not be said when planning inspection of the plane **??**, with state space reduced to 2 dimensions by constraining z, yaw, and pitch. The paths resulting from the augmented inspection planner are consistently longer for all number of max iterations. A possible explaination is that the path returned by the unaugmented algorithm 5.5 includes multiple infeasible viewpoints or connections between viewpoints. This is further backed up by the lack of viewpoints in the vicinity of the start and end positions. The results from this test more closely resemble the expected performance of the algorithm after the augmentations, as the state space appears to be more constrained and the resulting path is longer and has a less convincing visual.

Possible sources of errors

When investigating the source of the varying behavior and time for convergence given a different number of max iterations, the only possible source of this type of behavior is the smoothing parameter, defined in the code to only compile if the vehicle model selected is the fixed wing. As this part of the code compiles and runs without the parameter enabling the *ifdef* term being set, it is difficult to say if the experiments have been done with or without the extra functionality designed for a fixed-wing AUV. Luckily the smoothing algorithm improves path convergence as shown in chapter 5.2.4.

6.1.2 Convergence

There appears to be significant variance to the solution generated with 20 iterations of the planner. However, the length of the paths appears to be more consistent as the number of iterations increase.

The general behavior of the algorithm regardless of a number of iterations is a few seconds without much improvement, until the solution converges exponentially towards some value, stabilizes itself there, then given enough iterations will converge either asymptotically or exponentially towards a more optimal value.

6.1.3 Complexity

The augmented inspection planner spends roughly twice the amount of time as the unaugmented planner. This makes sense as the augmentation will call the inverse kinematics solver plugin multiple times each iteration. While the viewpoint feasibility criterion is not guaranteed to run, the TSP is solved multiple times each iteration. Resulting in multiple calls to the updated distance estimation function.

6.1.4 Simulation results

The augmented inspection planning algorithm was only able to make one plan resulting in a full trajectory by the Cartesian motion planner.

While most of the plans worked with the inverse kinematics motion planner(IKMP), only the plans generated by the augmented inspection planning algorithm produced plans that reliably executed without taking major detours from point to point.

If a motion planner like the IKMP is desired, then the augmentations are a definite improvement. However, the Cartesian motion planner showed considerably better accuracy and execution speed. The algorithm versions with the simple inverse kinematic check where the only ones where multiple viewpoints where considered unreachable 5.7. Even the unaugmented algorithm produced a path where every viewpoint was reachable. It is, therefore, possible that the emphasis on ensuring all viewpoints being feasible is a step in the wrong direction, as it appears adding the single inverse kinematics check made the robot performance worse when simulating the paths. The paths generated by the augmented algorithm(v4) had significantly better performance during simulation with the IKMP than the paths generated by the other algorithm versions. Meaning the measures intended to adapt the inspection plan for use with the UR5 worked to some degree. All augmentations seemed to decrease the total time spent executing the trajectory compared to the path generated by the unaugmented algorithm.

However attempting to simulate the path with the Cartesian motion planner, only the inspection path for the mount could successfully be made into a robot trajectory. Since the Cartesian motion planner appeared to be more suitable for inspection, the inspection planner is still not good enough.

6.1.5 Evaluation of inspection planning approach

The augmentations to the algorithm improved both path length and robot behaviour, at the cost of complexity. The results when testing the effectiveness of the individual improvements hints at viewpoint feasibility not being the main obstacle between the inspection planner and optimal motion plans.

Comparing the paths generated to the expected optimal path, the generated paths are similar but appears to be contain multiple redundant viewpoints. A proper implementation of an algorithm removing redundant viewpoints could have significant results.

The inspection planner assumes the end effector is able to move linearly between each viewpoint. So far the only measure taken to ensure path viability is to penalize movement between viewpoints where a straight line would pass through poses in which the manipulator has no feasible configurations, or the inverse kinematics problem cannot be solved. This appears to have little to no effect alone, as paths generated with only this measure contained arcing motions between several viewpoints 5.17. The intended effect of having viable linear paths between the viewpoints appears to fail, as no inspection path except the plan for the mount succeeded when planning motion with the Cartesian motion planner.

The distance evaluation extension proposed is intended as a measure to improve path quality, but still planning the optimal tour in tool space. However, it can be argued that the viability of each viewpoint but the optimal path between them is the problem. The motion produced by the Cartesian motion planner appears to be more accurate and predictable compared to the motion resulting from the IKMP approach. The inspection planning algorithm is already designed to output a linear trajectory in Cartesian space. A step towards planning for a manipulator would be to combine the Cartesian motion planner and the inspection planner. Assuming an algorithm iteratively improving a trajectory specified by ordered points in joint space is more suited for inspection with a manipulator.

To go from expressing the tour in tool space to expressing it in joint space, there needs to be an overhaul of the infrastructure of the inspection planner. As such a lot of the code would have to be rewritten, considering that altering the existing infrastructure is a laborious process. However the framework for placing the sensor seems to require minimal further improvement.

Next Best View vs Offline planner

It is worth noting that an offline pre computed approach is often referenced to in papers together with aerial or underwater drones when inspecting larger objects, but rarely when inspecting complex objects using a stationary robot. There could be many reasons to this, such as drones having motion and collision conveniently defined in cartesian coordinates as well. While inspection planning for a manipulator, you either have to consider model visibility while planning the trajectory in joint space, which might lead to non trivial work space constraints, when solving the constraint satisfaction problem. Or plan sensor poses and inspection path in cartesian space, and have measures to generate a valid and optimal joint trajectory from that.

For indoor inspection of complex objects, the focus is often construction of an assumed unavailable 3D model. Thus Next Best View and similar approaches are the most common occurrence in literature. As the NBV variants are greedy online algorithms the inspection is planned while scanning and the objective is often to minimize the number of scans required rather than optimizing the path between separate scans to minimize time.

For large structures, if the given placement of the structure fixed position is off by a few centimeters it is probably a minimal source of error. However if quality control requires millimeter precision, not being able to measure and adapt to object orientation and position could be a major hindrance. Measures to correct the plan while inspecting should be considered for further improvement of the approach taken in this thesis.

6.2 Evaluation of motion planning approaches

Two different approaches was made when planning robot motion. Both motion planning approaches where tested with the same trajectories and compared against each other.

6.2.1 Inverse Kinematics-based Motion Planner(IKMP)

The IKMP is a point to point motion planner, aiming to minimize movement of the robot between viewpoints.

A positive side to point to point based motion planning is that inspection paths can be simulated even if several of the viewpoints are in-feasible. However path quality is not guaranteed between viewpoints, as paths are either linear and close to optimal or arcing reaching the outer bounds of the workspace. The IKMPs strength is robustness and its weakness is path quality. One can assume that quality control is a repetitive task, as such the ability to simulate any given path is less important than the quality of the paths it is able to produce.

While stopping at each viewpoint to ensure at least 1 frame is captured there could be seen as a positive trait. Frequent stops and starts, goal overshoot, lower accuracy and longer total execution time for identical paths when compared to the Cartesian motion planning approach, are all strong arguments against continuing with this approach.

There are several possible reasons for the path quality inconsistencies. The point to point movement is planned by a planning algorithm from the OMPL, chosen automatically by MoveIt. For the UR5, the algorithm chosen is RRTConnect. RRTConnect is a

Rapidly-expanding Random Tree based algorithm, expanding one tree from each state and returning the first path found connecting the states. If the direct connection contains collisions or singular configurations, the returned path is likely to be along the side branches of the tree, resulting in large arcing motions even after post processing the path.

Attempting point to point planning with an asymptotically optimal point to point planner such as RRT* and significantly increasing available planning time, or multi-query planners such as PRM* may be a solution to improve path quality.

Possible sources of errors

The analytical function by Hawkins (2013) calculates inverse kinematics for a UR5 without any end effector. To get joint configurations placing the sensor at the desired pose, the IKMP must compensate for end effector offset before calling the solver.

Observing the paths from simulation 5.20 it appears the end-effector offset is not giving the correct values as there appears to be a near constant offset between the desired and logged trajectory, this is especially visible in the plots of robot motion compared to the desired trajectory for inspecting the plane. Which points towards an error in either setting up the end effector transformation in the URDF or an error calculating end effector offset.

The paths also show a significant error in orientation, which is present for both motion planning approaches. Meaning either the state logger has some errors in calculating forward kinematics unless neither of the motion planners is capable of producing configurations matching most orientations.

6.2.2 Cartesian trajectory planner

The Cartesian motion planner uses the Cartesian path service provided by MoveIt to produce paths linear between successive waypoints. It does this by interpolating between waypoints with a constant distance between successive points given by a step size parameter. Trajectory between points is planned in joint space, resulting in approximately linear movement of the end effector for sufficiently small step sizes. The algorithm appears to cancel path planning as soon as it encounters an obstacle or self-collision along the path, making it difficult to calculate a complete path. This was a significant problem when attempting to simulate the inspection paths. The cartesian planner excelled in accuracy and execution speed compared to the IKMP.

Both the cartesian trajectory planner and the inverse kinematics trajectory planner had significant errors regarding orientation of the end effector. This could possibly be because of inconsistencies between the analytic inverse kinematics, offset to compensate for the sensor location for the point to point planner. However both the cartesian trajectory planner and the robot state logger uses the kinematics plugin of the move group node. Meaning the orientation error can not be because of inconsistencies with urdf, or different implementations of kinematics solvers. Meaning the planner is either unable to create a trajectory following the specified pose with position and orientation, or there are significant numerical errors when calculating back and forth between cartesian space and joint space.

Descartes, dense planner

The attempted installation of the Descartes package was to test the functionality of the dense planner. To see if it would produce high quality linear multi-goal trajectories without the need of careful and rigorous tuning. However getting it to work with the UR5 took much more time than anticipated and further attempts of making a planner based on the functionality was aborted.

Given more time the performance of a possible planner should be investigated as a more robust and customize implementation of the cartesian path service.

6.2.3 Possible error with state logger

There where multiple problems with the robot state logger, the first being that the robot state publisher is not consistent with the order that joint states are published. When simulating in gazebo they will be published in alphabetical order, and the modern driver will publish the states in order from the robot base to the tool. Causing problems when logging end effector position.

There were consistent orientation errors during both simulation and experiments with the UR5. The robot state logger is possibly returning the wrong orientations consistently or neither of the motion planning approaches manage to accurately control orientation.

Error with URDF

There was an error in the robot URDF during the experiments. The line specifying the origin of the sensor was missing. As the tooltip was being specified as a joint 1 cm outwards from the origin of the sensor link, the URDF error resulted in the tooltip being positioned 1 cm out from the tip of the third link of the robot wrist instead of 5.5 cm out and 3.5 cm up. This is a possible explanation of the constant position errors of the IKMP. The Cartesian path planner and the robot state logger perform kinematic calculations based on the URDF, meaning an error in the URDF would not necessarily lead to an error between logged state and desired state.

6.2.4 Comparison

Both of the motion planning approaches appear to have trouble controlling sensor orientation. It is difficult to guarantee that a given part of the model has been observed when the sensor position may not satisfy the visibility criterion. Thus at the present time both motion planning approaches need improvement.

The improvements of the inspection planning algorithm, while improving performance during simulations, does not guarantee optimal traversal between each viewpoint. The cartesian motion planner fail to execute most inspection plans, and the IKMP still exhibits cases of unpredictable movement.

While both methods are capable of moving the robot, the real problem appears to be that planning the inspection trajectory in tool space is not optimal. Rewriting the inspection planner infrastructure in order to perform point to point planning in joint space should be attempted as it also has the benefits of cleaning up code that is not supposed to compile.

6.3 Point cloud acquisition

The SR300 had problems observing both the cube and the camera mount. Brightening the cube with red tape was attempted but did not improve the result significantly. The lighting conditions were not ideal, and the sensor seemed to struggle with dark surfaces. Even though it managed to capture most of a dark cup under different lighting conditions, as shown in figure 1.4.

While the 3D sensor captures at 30 frames per second, the system only managed to capture frames at an approximately 1.4 fps when logging the data. This could be a problem if the manipulator is moving faster than 1 viewpoint per second. Due to the loss of measurements from multiple viewpoints. The bottleneck of the frame logging is writing the data to file. Waiting with writing to file to after the scan is complete should significantly increase the number of frames being captured during execution.

There was no success with the transformation of the point cloud. This is either caused by faults in implementation, a misunderstanding of how the RGBD-nodelet processes the point cloud data with respect to transformation data or simply error orientation in the state data from the robot state logger.

6.3.1 Evaluation

The sensor seems to struggle with dark and reflective surfaces. While it is probably not up to industry standards, it is worth attempting to scan with better lighting to check if it improves performance, before discarding the sensor. The sensor has multiple upsides such as budget, size and driver support. While it is probably unsuitable for a commercial product it should perform well for research purposes.

As the point cloud transformations where not successful point cloud reconstruction will probably have to rely on ICP or similar algorithms for aligning the clouds correctly. If the goal is to reconstruct the 3D models.

Chapter 7

Conclusion

7.1 Closing statement on the proposed inspection planner

The proposed inspection planning augmentation greatly increased the path quality when simulating with the IKMP, while also improving path length when planning in 5 dimensions.

While there is a case for continuing to extend the algorithm, while planning the trajectory in tool space, there is significant benefits to combining them. Such as the possibility to re-plan problem viewpoints, the tour metric being the estimated length of the tour in joint space instead of an estimate based on sensor velocity and a cartesian trajectory, and therefore matching the resulting length 5.7 of the inspection path.

The augmentation appear to significantly improve manipulator trajectory quality, however the inspection planning and trajectory will benefit greatly by being done cooperatively.

7.2 Closing statement on the proposed motion planners

The Cartesian planner is accurate with regards to position, but difficult to make work with the current inspection planning algorithm, as planning fails to due to infeasibility along the path.

IKMP has steady improvements with the augmentation of the algorithm, and the low accuracy may be because of mistakes when configuring robot URDF and calculating tool center point offset. The frequent presence of arcing motions between waypoints and overshoot makes the approach inferior to the Cartesian motion planner.

Integrating Cartesian path functionality in the inspection planner appears to be the most promising solution. Either by using the functionality by MoveIt, or by using the dense planner from the Descartes package to plan a trajectory in joint space. Further augmentation of the inspection planner to guarantee the inspection plan being able to form a complete trajectory using the Cartesian motion planning service is an alternative approach.

7.3 Future work

For improving the automatic 3D scanning, the integration with motion planning must be drastically improved, either by augmenting the inspection planner to output plans that are feasible for the Cartesian motion planner or by rewriting the inspection planning algorithm to output a robot trajectory instead of a series of viewpoints.

The sensor had difficulties observing both the cube and the camera mount, a possible solution may be researching alternative sensors or improving the lighting conditions.

The simple transformation of the point clouds did not work as intended, implementing a fix to this as well as an ICP algorithm or similar measure to align the point clouds may be necessary to reconstruct models from the sensor measurements.

Based on the computational limitation of logging point cloud data implementing online CAD model reconstruction or drastically optimizing the logging of point cloud data is necessary. As capturing the data at under 2 fps is not sufficient if the goal is to increase scanning performance.

The point clouds returned had data with visible artifacts 5.30. So filtering to compensate for sensor noise and other artifacts is necessary.

- Rewrite inspection planner infrastructure to perform point to point planning in joint space with a cartesian path planner.
- Identify source of orientation error and fix it.
- Implement path post processing algorithm for removing redundant viewpoints.
- Look into Redundant Roadmap approach of coverage path planning.
- Look into making a working motion planner based on the descartes package
- Improve point cloud acquisition, by either acquiring a better sensor or by improved lighting
- Implement working algorithm for aligning the point clouds in a fixed frame.
- Implement point cloud filtering algorithms to remove artifacts from measurements
- Implement real-time online model reconstruction
- Investigate ways of providing online feedback to the inspection path to increase inspection quality.

Bibliography

Akbaripour, H., Masehian, E., 2016. Semi-lazy probabilistic roadmap: a parameter-tuned, resilient and robust path planning method for manipulator robots. The International Journal of Advanced Manufacturing Technology, 1–30.

Andersen, T. T., 2015. Optimizing the universal robots ros driver. Tech. rep., Technical University of Denmark, Department of Electrical Engineering.
URL `http://orbit.dtu.dk/en/publications/optimizing-the-universal-robots-ros-driver(20dde139-7e87-4552-8658-db.html`

Beeson, P., Ames, B., November 2015. TRAC-IK: An open-source library for improved solving of generic inverse kinematics. In: Proceedings of the IEEE RAS Humanoids Conference. Seoul, Korea.

Bergström, P., 2016. Reliable updates of the transformation in the iterative closest point algorithm. Computational Optimization and Applications 63 (2), 543–557.
URL `http://dx.doi.org/10.1007/s10589-015-9771-3`

Bergström, P., Edlund, O., 2016. Robust registration of surfaces using a refined iterative closest point algorithm with a trust region approach. Numerical Algorithms, 1–25.
URL `http://dx.doi.org/10.1007/s11075-016-0170-3`

Bircher, A., Alexis, K., Burri, M., Oettershagen, P., Omari, S., Mantel, T., Siegwart, R., 2015a. Structural inspection path planning via iterative viewpoint resampling with application to aerial robotics. In: Robotics and Automation (ICRA), 2015 IEEE International Conference on. IEEE, pp. 6423–6430.

Bircher, A., Kamel, M., Alexis, K., Burri, M., Oettershagen, P., Omari, S., Mantel, T., Siegwart, R., 2015b. Three-dimensional coverage path planning via viewpoint resampling and tour optimization for aerial robots. Autonomous Robots, 1–20.

Cappelletto, E., Zanuttigh, P., Cortelazzo, G. M., 2013. Handheld scanning with 3d cameras. In: Multimedia Signal Processing (MMSP), 2013 IEEE 15th International Workshop On. IEEE, pp. 367–372.

Englot, B., Hover, F., 2011. Planning complex inspection tasks using redundant roadmaps. In: Proc. Int. Symp. Robotics Research.

Englot, B., Hover, F., 2017. Planning complex inspection tasks using redundant roadmaps. In: Robotics Research. Springer, pp. 327–343.

Englot, B., Hover, F. S., 2012. Sampling-based coverage path planning for inspection of complex structures.

Erdős, G., Kardos, C., Kemény, Z., Kovács, A., Váncza, J., 2016. Process planning and offline programming for robotic remote laser welding systems. International Journal of Computer Integrated Manufacturing 29 (12), 1287–1306.

Ferreau, H. J., Kirches, C., Potschka, A., Bock, H. G., Diehl, M., 2014. qpoases: A parametric active-set algorithm for quadratic programming. Mathematical Programming Computation 6 (4), 327–363.

Galceran, E., Carreras, M., 2013. A survey on coverage path planning for robotics. Robotics and Autonomous Systems 61 (12), 1258–1276.

Gasparetto, A., Zanotto, V., 2007. A new method for smooth trajectory planning of robot manipulators. Mechanism and machine theory 42 (4), 455–471.

Hawkins, K. P., 2013. Analytic inverse kinematics for the universal robots ur-5/ur-10 arms. Tech. rep., Georgia Institute of Technology.

Helsgaun, K., 2000. An effective implementation of the lin–kernighan traveling salesman heuristic. European Journal of Operational Research 126 (1), 106–130.

Jimenez, P. A., Shirinzadeh, B., Nicholson, A., Alici, G., 2007. Optimal area covering using genetic algorithms. In: Advanced intelligent mechatronics, 2007 IEEE/ASME international conference on. IEEE, pp. 1–5.

Jolliffe, I., 2002. Principal component analysis. Wiley Online Library.

Karaman, S., Frazzoli, E., 2011. Sampling-based algorithms for optimal motion planning. The international journal of robotics research 30 (7), 846–894.

Karaszewski, M., Adamczyk, M., Sitnik, R., 2016. Assessment of next-best-view algorithms performance with various 3d scanners and manipulator. ISPRS Journal of Photogrammetry and Remote Sensing 119, 320–333.

Krainin, M., Curless, B., Fox, D., 2011. Autonomous generation of complete 3d object models using next best view manipulation planning. In: Robotics and Automation (ICRA), 2011 IEEE International Conference on. IEEE, pp. 5031–5037.

Kriegel, S., Rink, C., Bodenmüller, T., Suppa, M., 2015. Efficient next-best-scan planning for autonomous 3d surface reconstruction of unknown objects. Journal of Real-Time Image Processing 10 (4), 611–631.

Kuffner, J. J., LaValle, S. M., 2000. Rrt-connect: An efficient approach to single-query path planning. In: Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on. Vol. 2. IEEE, pp. 995–1001.

Marton, Z. C., Rusu, R. B., Beetz, M., May 12-17 2009. On Fast Surface Reconstruction Methods for Large and Noisy Datasets. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA). Kobe, Japan.

Rusu, R. B., Cousins, S., 2011. 3d is here: Point cloud library (pcl). In: Robotics and Automation (ICRA), 2011 IEEE International Conference on. IEEE, pp. 1–4.

Saha, M., Roughgarden, T., Latombe, J.-C., Sánchez-Ante, G., 2006. Planning tours of robotic arms among partitioned goals. The International Journal of Robotics Research 25 (3), 207–223.

Seines, M. M., December 2016. Automatic 3d inspection with robot manipulator.

Tangelder, J. W., Veltkamp, R. C., 2003. Polyhedral model retrieval using weighted point sets. International journal of image and graphics 3 (01), 209–229.

Tuzikov, A. V., Roerdink, J. B., Heijmans, H. J., 2000. Similarity measures for convex polyhedra based on minkowski addition. Pattern Recognition 33 (6), 979–995.

Vasquez-Gomez, J. I., Sucar, L. E., Murrieta-Cid, R., 2014. View planning for 3d object reconstruction with a mobile manipulator robot. In: 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE, pp. 4227–4233.

Wu, S., Sun, W., Long, P., Huang, H., Cohen-Or, D., Gong, M., Deussen, O., Chen, B., 2014. Quality-driven poisson-guided autoscanning. ACM Transactions on Graphics 33 (6).

Yang, K., Sukkarieh, S., 2010. An analytical continuous-curvature path-smoothing algorithm. IEEE Transactions on Robotics 26 (3), 561–568.

Appendix

7.4 ROS terminology

7.4.1 Launch files

ROS works as a framework for managing nodes, a parameter server for shared information and manage subscribers and publishers for topics providing information. A node can either be run directly if it does not require external parameters or similar functionality. A node is a process that performs computation, and communicate with one another through topics, services, and a parameter server.

To run a node directly the rosrun command is used. A node may require certain parameters to be loaded into the parameter server, or certain services to access. Therefore, while rosrun will start running the node, it may fail or have segmentation faults because of missing parameters or services. It is therefore often required to run or launch different nodes in a specific order, or pass it specific arguments.

- rosrun package_ name executable $(argument1 := value1 \quad argument2 := ...)$

If a service or process is comprised of a number of different nodes that has to be launched in some specific order, a set of parameters needs to be uploaded to the paramter server. Then this is specified in a $.launch$ file that helps start the nodes in the correct order, with the correct arguments.

- roslaunch package service.launch $(argument1 := value1 \quad argument2 := ...)$

Inspection planner example

To run the algorithm the planner is launched from a launch file as a ROS service in a terminal window.

- roslaunch koptplanner ur5.launch

The user then runs a client specifying the mesh that needs inspection, as well as parameters not related to the choice of craft (Maximal velocity, and camera pitch). The client is assumed to be part of the request package also found in the source. A client is launched from terminal by the command.

- rosrun request ”scenario specific executable” _ mesh:=”meshname.stl”

7.4.2 Using Rviz

Rviz is the default ROS gui. It is a 3D visualization tool that can be set up to visualize most standard topics. Rviz is used to visualize both path and mesh, viewpoints are represented as arrows with a light blue connecting path representing the solution of the TSP of the given set of points. The provided mesh is given as set of green triangles.

- rosrun rviz rviz

To visualize a robot rviz can be launched with a preset configuration visualizing the robot and its trajectory.

- roslaunch robot_ moveit_ config moveit_ rviz.launch config:=true

7.4.3 Topics and subscribers

To communicate between nodes in you use topics. Topics are messages of a spesific class. for instance a spesific class for messages containing information of joint states of the robot.

All messages have a standard header containing timing info, frame id and a sequence number.

To send a topic, initiate a publisher and this will announce to the ros master that somebody is publishing on the topic, and any other node can make a subscriber receiving this information. Topics are named after the name of the node, followed by the specified name.

example code for advertising a topic
ros::init(argv, argc, ”node name”);
ros::nodeHandle nh;
ros::Publisher pub;
nh.advertise¡”message type”¿ (”topic name”, buffer size);

The above code will result in a topic for *”message type”* data to be available under */”node name”/”topic name”* visible to the ROS master allowing other nodes to subscribe to it.

To subscribe to a node, make a subscriber object and specify a callback function used to process the message.

example code for subscribing to a topic
void topicCallback(const ”message type” & msg);
ros::init(argv, argc, ”node name2”);
ros::nodeHandle nh;
ros::Subscriber sub = nh.subscribe(”/node name/topic name”, buffer size, topicCallback);

ROS will automatically buffer overload, however it is possible to specify if it is desirable to keep the oldest messages received or keep the newest message.

7.5 Configuring the robot

7.5.1 Configuring robot URDF

Normal URDF

```xml
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro"
       name="ur5" >

  <!-- this file contains necessary macros to allow visualization and simulation in gazebo -->
  <xacro:include filename="$(find ur_description)/urdf/common.gazebo.xacro" />

  <!-- ur5: ur5.urdf.xacro is the urdf found in the ur_description package -->
  <xacro:include filename="$(find ur5_with_sr300_support)/urdf/ur5.urdf.xacro" />

  <!-- arm -->
  <xacro:ur5_robot prefix="" joint_limited="false"/>

  <link name="world" />

  <joint name="world_joint" type="fixed">
    <parent link="world" />
    <child link = "base_link" />
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
  </joint>

  <! -- visualize the sensor -->
  <link name="sensor">
    <visual>
        <origin xyz="0.0 0.0 0.0" rpy="0 0.0 0.0" />
        <geometry>
            <box size="0.03 0.12 0.03"/>
        </geometry>
        <material name="Grey" />
    </visual>
    <collision>
        <origin xyz="0.0 0.0 0.0" rpy="0 0 0" />
        <geometry>
            <box size="0.03 0.12 0.03"/>
        </geometry>
    </collision>
  </link>

  <! -- symbolic link representing tip of sensor -->
  <link name="tool_tip" />

  <! -- Specify joint representing origin of sensor -->
  <joint name="tool_to_sensor" type="fixed">
    <parent link="ee_link"/>
    <child link="sensor"/>
    <origin xyz="0.045 0.0 0.035" rpy="0 0.0 0.0" /> <!--
       THIS LINE WAS MISSING -->
  </joint>

 <! -- position of depth camera on the sensor relative to
    COM -->
 <joint name="sensor_to_tip" type="fixed">
   <parent link="sensor"/>
   <child link="tool_tip"/>
   <origin xyz="0.01 0.0 0.0" rpy="0 0.0 0" />
 </joint>

</robot>
```

Joint limited URDF

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro"
       name="ur5" >

  <!-- this file contains necessary macros to allow
      visualization and simulation in gazebo -->
  <xacro:include filename="$(find ur_description)/urdf/
      common.gazebo.xacro" />

  <!-- ur5: ur5.urdf.xacro is the urdf found in the
      ur_description package -->
  <xacro:include filename="$(find ur5_with_sr300_support)/
      urdf/ur5.urdf.xacro" />

  <!-- arm -->
  <xacro:ur5_robot prefix="" joint_limited="true"/>

  <link name="world" />

  <joint name="world_joint" type="fixed">
    <parent link="world" />
    <child link = "base_link" />
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
  </joint>

  <! -- visualize the sensor -->
  <link name="sensor">
    <visual>
        <origin xyz="0.0 0.0 0.0" rpy="0 0.0 0.0" />
        <geometry>
            <box size="0.03 0.12 0.03"/>
        </geometry>
        <material name="Grey" />
    </visual>
    <collision>
        <origin xyz="0.0 0.0 0.0" rpy="0 0 0" />
        <geometry>
            <box size="0.03 0.12 0.03"/>
        </geometry>
    </collision>
  </link>

  <! -- symbolic link representing tip of sensor -->
  <link name="tool_tip" />

  <! -- Specify joint representing origin of sensor -->
  <joint name="tool_to_sensor" type="fixed">
    <parent link="ee_link"/>
    <child link="sensor"/>
    <origin xyz="0.045 0.0 0.035" rpy="0 0.0 0.0" /> <!--
        THIS LINE WAS MISSING -->
  </joint>

 <! -- position of depth camera on the sensor relative to
     COM -->
 <joint name="sensor_to_tip" type="fixed">
   <parent link="sensor"/>
   <child link="tool_tip"/>
   <origin xyz="0.01 0.0 0.0" rpy="0 0.0 0" />
 </joint>

</robot>
```

Default robot description ur5.urdf.xacro can be fount at https://github.com/ros-industrial/universal_robot/blob/kinetic-devel/ur_description/urdf/ur5.urdf.xacro

7.5.2 Configurating scripts for loading custom robot into Gazebo

First a script for loading the robot description into the parameter server needs to be made.

```
<?xml version="1.0"?>
<launch>
  <arg name="limited" default="false"/>

  <param unless="$(arg limited)" name="robot_description"
      command="$(find xacro)/xacro --inorder '\$(find
      ur5_with_sr300_support)/urdf/ur5_robot.urdf.xacro'" />
  <param if="\$(arg limited)" name="robot_description"
      command="\$(find xacro)/xacro --inorder '\$(find
      ur5_with_sr300_support)/urdf/ur5_joint_limited_robot.
      urdf.xacro'" />
</launch>
```

The script for starting gazebo with the custom robot can be copied from the ur_ gazebo package and slightly altered to upload the custom robot instead of the default robot.

```
<?xml version="1.0"?>
<launch>
  <arg name="limited" default="false"/>
  <arg name="paused" default="false"/>
  <arg name="gui" default="true"/>

  <!-- startup simulated world -->
  <include file="\$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" default="worlds/empty.world"/>
    <arg name="paused" value="\$(arg paused)"/>
    <arg name="gui" value="\$(arg gui)"/>
  </include>

  <!-- send robot urdf to param server -->
  <!-- altered to find location of new upload file and launch it -->
  <include file="\$(find ur5_with_sr300_gazebo)/launch/ur5_with_sr300_upload.launch">
    <arg name="limited" value="\$(arg limited)"/>
  </include>

  <!-- push robot_description to factory and spawn robot in gazebo -->
  <node name="spawn_gazebo_model" pkg="gazebo_ros" type="spawn_model" args="-urdf -param robot_description -model robot -z 1.1" respawn="false" output="screen" />

  <include file="\$(find ur_gazebo)/launch/controller_utils.launch"/>

  <rosparam file="\$(find ur_gazebo)/controller/arm_controller_ur5.yaml" command="load"/>
  <node name="arm_controller_spawner" pkg="controller_manager" type="controller_manager" args="spawn arm_controller" respawn="false" output="screen"/>

</launch>
\end{lstlistings}

Create a wrapper for launching the move group node to allow simply switching communication between simulation in gazebo and a robot driver given the "sim" argument.

\begin{lstlisting}[language=XML]
<?xml version="1.0"?>
<launch>
  <arg name="sim" default="false" />
  <arg name="debug" default="false" />
        <arg name="limited" default="false" />

  <!-- Remap follow_joint_trajectory -->
  <remap if="\$(arg sim)" from="/follow_joint_trajectory"
     to="/arm_controller/follow_joint_trajectory"/>

  <!-- Launch moveit -->
  <include file="\$(find ur5_with_sr300_moveit_config)/
     launch/move_group.launch">
                  <arg name="limited" default= "\$(arg
                     limited)" />
                  <arg name="debug" default="\$(arg debug)" /
                     >
  </include>
</launch>
```

Fixes to auto generate config files

The moveit controller manager launch file will not be generated correctly by default. Open the moveit config folder generated earlier and modify the file called: robot_ moveit_ controller_ manager.launch.xml

```
<launch>
  <rosparam file="\$(find ur5_with_sr300_moveit_config)/
     config/controllers.yaml"/>
  <param name="use_controller_manager" value="false"/>
  <param name="trajectory_execution/
     execution_duration_monitoring" value="false"/>
  <param name="moveit_controller_manager" value="
     moveit_simple_controller_manager/
     MoveItSimpleControllerManager"/>
</launch>
```

controllers.yaml

```
controller_list:
  - name: ""
    action_ns: follow_joint_trajectory
    type: FollowJointTrajectory
    joints:
      - shoulder_pan_joint
      - shoulder_lift_joint
      - elbow_joint
      - wrist_1_joint
      - wrist_2_joint
      - wrist_3_joint
```

The launch files for move group, will not automaticly support loading one of two URDF files and need to be changed by hand to do so. move_ group.launch

```
<?xml version="1.0"?>
<launch>

        <arg name="limited" default="false"/>

  <include file="$(find ur5_with_sr300_moveit_config)/
     launch/planning_context.launch" >
                <arg name="limited" value="$(arg limited)"
                   />
  </include>

  <!-- GDB Debug Option -->
  <arg name="debug" default="false" />
  <arg unless="$(arg debug)" name="launch_prefix" value=""
     />
  <arg     if="$(arg debug)" name="launch_prefix"
           value="gdb -x $(find
              ur5_with_sr300_moveit_config)/launch/
              gdb_settings.gdb --ex run --args" />

  <!-- Verbose Mode Option -->
  <arg name="info" default="$(arg debug)" />
  <arg unless="$(arg info)" name="command_args" value="" />
  <arg     if="$(arg info)" name="command_args" value="--
     debug" />


  <!-- move_group settings -->
  <arg name="allow_trajectory_execution" default="true"/>
  <arg name="fake_execution" default="false"/>
  <arg name="max_safe_path_cost" default="1"/>
  <arg name="jiggle_fraction" default="0.05" />
  <arg name="publish_monitored_planning_scene" default="
     true"/>

  <!-- Planning Functionality -->
  <include ns="move_group" file="$(find
     ur5_with_sr300_moveit_config)/launch/planning_pipeline
     .launch.xml">
    <arg name="pipeline" value="ompl" />
  </include>

  <!-- Trajectory Execution Functionality -->
  <include ns="move_group" file="$(find
     ur5_with_sr300_moveit_config)/launch/
    trajectory_execution.launch.xml" if="$(arg allow_trajectory_execution)">
    <arg name="moveit_manage_controllers" value="true" />
    <arg name="moveit_controller_manager" value="ur5" unless="$(arg fake_execution)"/>
    <arg name="moveit_controller_manager" value="fake" if="$(arg fake_execution)"/>
  </include>

  <!-- Sensors Functionality -->
  <include ns="move_group" file="$(find ur5_with_sr300_moveit_config)/launch/sensor_manager.launch.xml" if="$(arg allow_trajectory_execution)">
    <arg name="moveit_sensor_manager" value="ur5" />
  </include>

  <!-- Start the actual move_group node/action server -->
  <node name="move_group" launch-prefix="$(arg launch_prefix)" pkg="moveit_ros_move_group" type="move_group" respawn="false" output="screen" args="$(arg command_args)">
    <!-- Set the display variable, in case OpenGL code is used internally -->
    <env name="DISPLAY" value="$(optenv DISPLAY :0)" />

    <param name="allow_trajectory_execution" value="$(arg allow_trajectory_execution)"/>
    <param name="max_safe_path_cost" value="$(arg max_safe_path_cost)"/>
    <param name="jiggle_fraction" value="$(arg jiggle_fraction)" />

    <!-- load these non-default MoveGroup capabilities -->
    <!--
    <param name="capabilities" value="
                  a_package/AwsomeMotionPlanningCapability
                  another_package/GraspPlanningPipeline
                  " />
    -->

    <!-- inhibit these default MoveGroup capabilities -->
    <!--
    <param name="disable_capabilities" value="
                  move_group/MoveGroupKinematicsService
                  move_group/ClearOctomapService
```

```
                     " />
    -->

    <!-- Publish the planning scene of the physical robot
       so that rviz plugin can know actual robot -->
    <param name="planning_scene_monitor/
       publish_planning_scene" value="$(arg
       publish_monitored_planning_scene)" />
    <param name="planning_scene_monitor/
       publish_geometry_updates" value="$(arg
       publish_monitored_planning_scene)" />
    <param name="planning_scene_monitor/
       publish_state_updates" value="$(arg
       publish_monitored_planning_scene)" />
    <param name="planning_scene_monitor/
       publish_transforms_updates" value="$(arg
       publish_monitored_planning_scene)" />
  </node>

</launch>
```

planning_ context.launch

```
<?xml version="1.0"?>
<launch>
  <!-- By default we do not overwrite the URDF. Change the
     following to true to change the default behavior -->
  <arg name="load_robot_description" default="false"/>
        <arg name="limited" default="false" />

  <!-- The name of the parameter under which the URDF is
     loaded -->
  <arg name="robot_description" default="robot_description"
     />

  <!-- Load universal robot description format (URDF) -->
  <group if="$(arg load_robot_description)">
    <param unless="$(arg limited)" name="$(arg
       robot_description)" command="$(find xacro)/xacro.py
       '$(find ur5_with_sr300_support)/urdf/ur5_robot.urdf.
       xacro'" />
    <param if="$(arg limited)" name="$(arg
       robot_description)" command="$(find xacro)/xacro.py
       '$(find ur5_with_sr300_support)/urdf/
       ur5_joint_limited_robot.urdf'" />
  </group>

  <!-- The semantic description that corresponds to the
    URDF -->
  <param name="$(arg robot_description)_semantic" textfile=
    "$(find ur5_with_sr300_moveit_config)/config/ur5.srdf"
     />

  <!-- Load updated joint limits (override information from
     URDF) -->
  <group ns="$(arg robot_description)_planning">
    <rosparam command="load" file="$(find
       ur5_with_sr300_moveit_config)/config/joint_limits.
       yaml"/>
  </group>

  <!-- Load default settings for kinematics; these settings
     are overridden by settings in a node's namespace -->
  <group ns="$(arg robot_description)_kinematics">
    <rosparam command="load" file="$(find
       ur5_with_sr300_moveit_config)/config/kinematics.yaml
       "/>
  </group>

</launch>
```

Changing OMPL settings

Changing setting to the point to point planning algorithms from the Open Motion Planning Library is done by altering the OMPLplanning.yaml configuration file in the custom moveit generation folder.

The planning algorithm is selected automatically. So to force a given planner simply comment out all other configurations.

```
planner_configs:
  SBLkConfigDefault:
    type: geometric::SBL
    range: 0.0  # Max motion added to tree. ==>
       maxDistance_ default: 0.0, if 0.0, set on setup()
  ESTkConfigDefault:
    type: geometric::EST
    range: 0.0  # Max motion added to tree. ==>
       maxDistance_ default: 0.0, if 0.0 setup()
    goal_bias: 0.05  # When close to goal select goal, with
        this probability. default: 0.05
  LBKPIECEkConfigDefault:
    type: geometric::LBKPIECE
    range: 0.0  # Max motion added to tree. ==>
       maxDistance_ default: 0.0, if 0.0, set on setup()
    border_fraction: 0.9  # Fraction of time focused on
       boarder default: 0.9
    min_valid_path_fraction: 0.5  # Accept partially valid
       moves above fraction. default: 0.5
  BKPIECEkConfigDefault:
    type: geometric::BKPIECE
    range: 0.0  # Max motion added to tree. ==>
       maxDistance_ default: 0.0, if 0.0, set on setup()
    border_fraction: 0.9  # Fraction of time focused on
       boarder default: 0.9
    failed_expansion_score_factor: 0.5  # When extending
       motion fails, scale score by factor. default: 0.5
    min_valid_path_fraction: 0.5  # Accept partially valid
       moves above fraction. default: 0.5
  KPIECEkConfigDefault:
    type: geometric::KPIECE
    range: 0.0  # Max motion added to tree. ==>
       maxDistance_ default: 0.0, if 0.0, set on setup()
    goal_bias: 0.05  # When close to goal select goal, with
        this probability. default: 0.05
    border_fraction: 0.9  # Fraction of time focused on
       boarder default: 0.9 (0.0,1.]
    failed_expansion_score_factor: 0.5  # When extending
       motion fails, scale score by factor. default: 0.5
    min_valid_path_fraction: 0.5  # Accept partially valid
       moves above fraction. default: 0.5
  RRTkConfigDefault:
    type: geometric::RRT
    range: 0.0  # Max motion added to tree. ==>
       maxDistance_ default: 0.0, if 0.0, set on setup()
    goal_bias: 0.05  # When close to goal select goal, with
        this probability? default: 0.05
  RRTConnectkConfigDefault:
    type: geometric::RRTConnect
    range: 0.0  # Max motion added to tree. ==>
       maxDistance_ default: 0.0, if 0.0, set on setup()
  RRTstarkConfigDefault:
    type: geometric::RRTstar
    range: 0.0  # Max motion added to tree. ==>
       maxDistance_ default: 0.0, if 0.0, set on setup()
    goal_bias: 0.05  # When close to goal select goal, with
        this probability? default: 0.05
    delay_collision_checking: 1  # Stop collision checking
```

```
        as soon as C-free parent found. default 1
  TRRTkConfigDefault:
    type: geometric::TRRT
    range: 0.0  # Max motion added to tree. ==> maxDistance_ default: 0.0, if 0.0, set on setup()
    goal_bias: 0.05  # When close to goal select goal, with this probability? default: 0.05
    max_states_failed: 10  # when to start increasing temp. default: 10
    temp_change_factor: 2.0  # how much to increase or decrease temp. default: 2.0
    min_temperature: 10e-10  # lower limit of temp change. default: 10e-10
    init_temperature: 10e-6  # initial temperature. default: 10e-6
    frountier_threshold: 0.0  # dist new state to nearest neighbor to disqualify as frontier. default: 0.0 set in setup()
    frountierNodeRatio: 0.1  # 1/10, or 1 nonfrontier for every 10 frontier. default: 0.1
    k_constant: 0.0  # value used to normalize expresssion. default: 0.0 set in setup()
  PRMkConfigDefault:
    type: geometric::PRM
    max_nearest_neighbors: 10  # use k nearest neighbors. default: 10
  PRMstarkConfigDefault:
    type: geometric::PRMstar
manipulator:
  planner_configs:
    - SBLkConfigDefault
    - ESTkConfigDefault
    - LBKPIECEkConfigDefault
    - BKPIECEkConfigDefault
    - KPIECEkConfigDefault
    - RRTkConfigDefault
    - RRTConnectkConfigDefault
    - RRTstarkConfigDefault
    - TRRTkConfigDefault
    - PRMkConfigDefault
    - PRMstarkConfigDefault
```

7.6 Results

7.6.1 Inspection plans

Optimal path plane, unmodified algorithm

Figure 7.1 Result after running the algorithm with locked z, pitch and yaw, optimal solution after 100 iterations

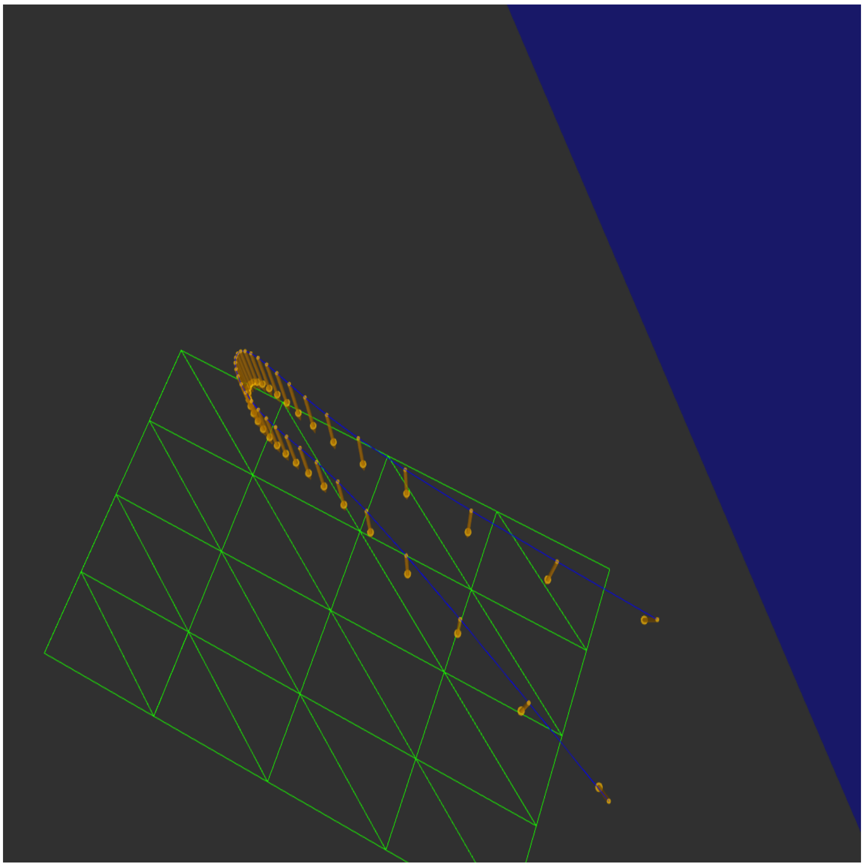


Figure 7.2 Algorithm v4 optimal solution

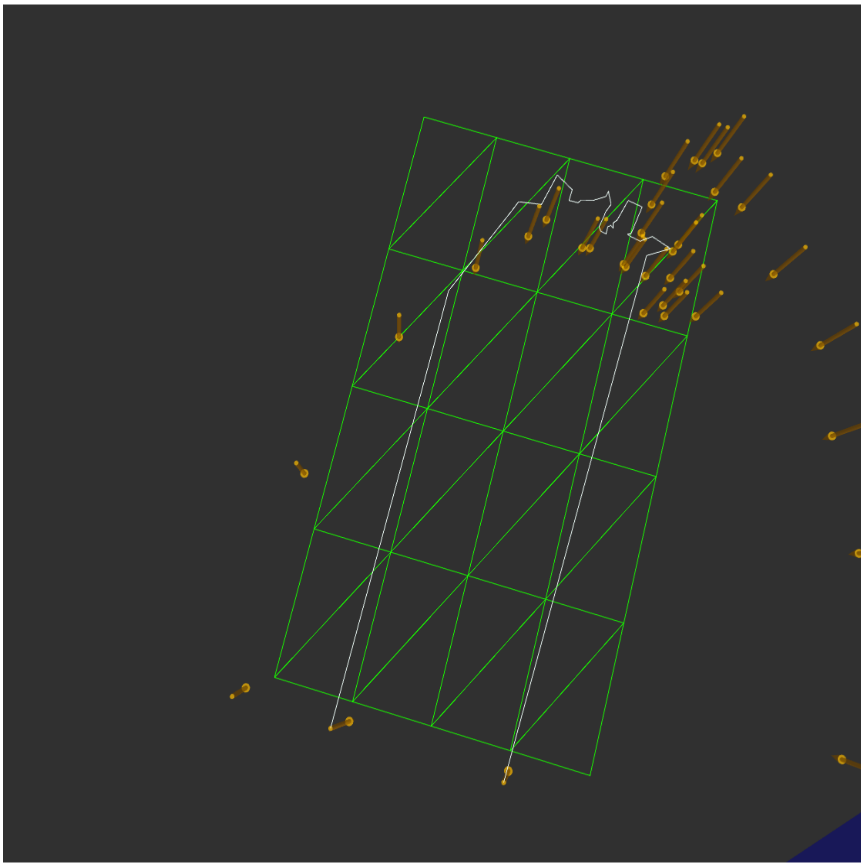
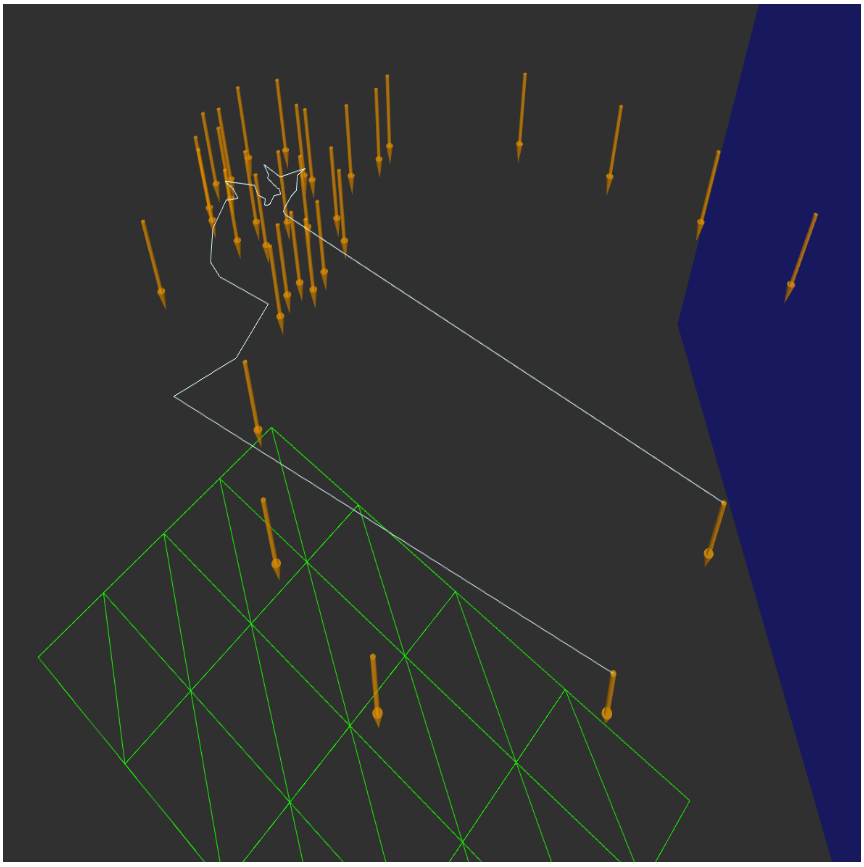
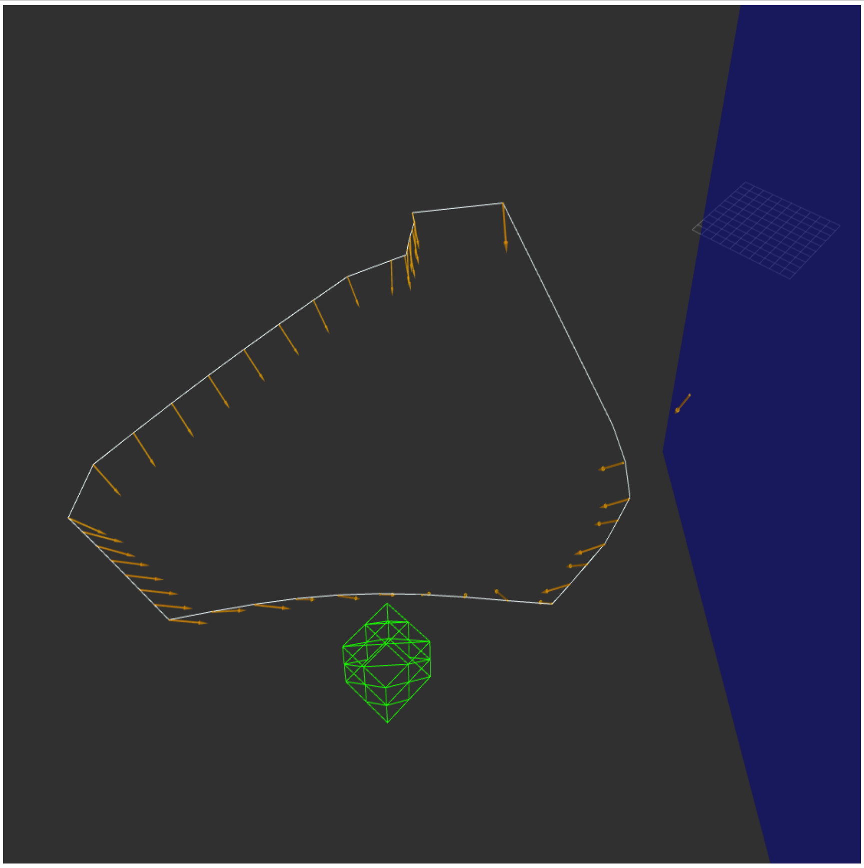


Figure 7.3 Algorithm v4 optimal solutons



Optimal path cube, unmodified algorithm

Figure 7.4 Result after running the algorithm as presented in my paper, 50 iterations, optimal solution



Optimal path cube augmented algorithm

Figure 7.5 Optimal path found with augmented algorithm running 100 iterations, resulting in a path of length 9.99

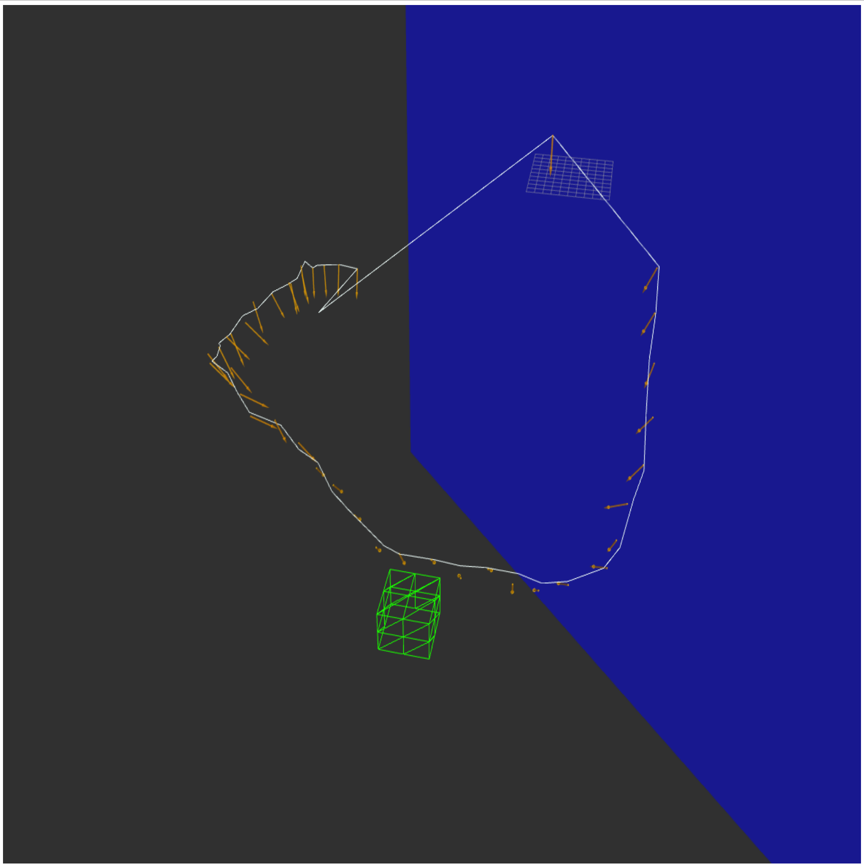
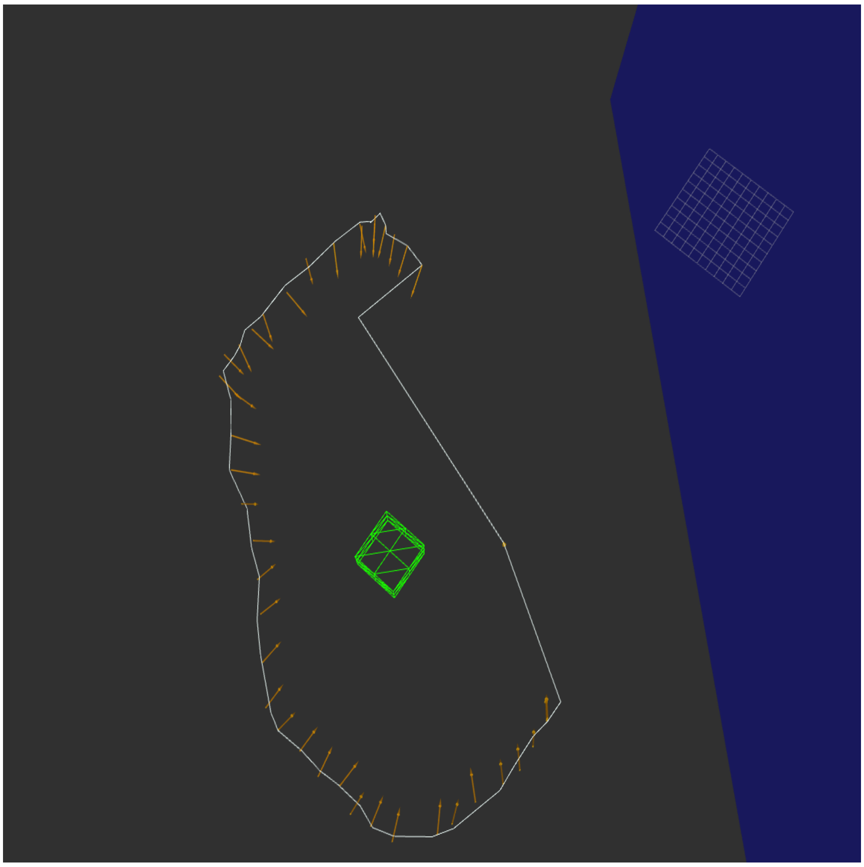


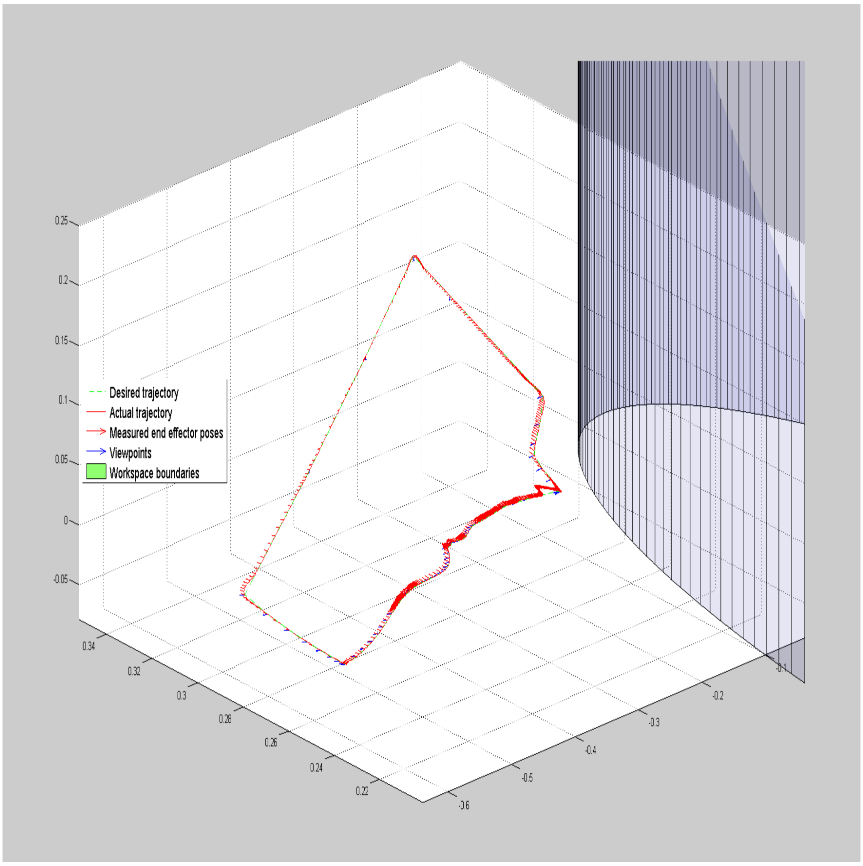
Figure 7.6 Optimal path found with augmented algorithm running 100 iterations, resulting in a path of length 9.99



7.6.2 Robot trajectories, simulation

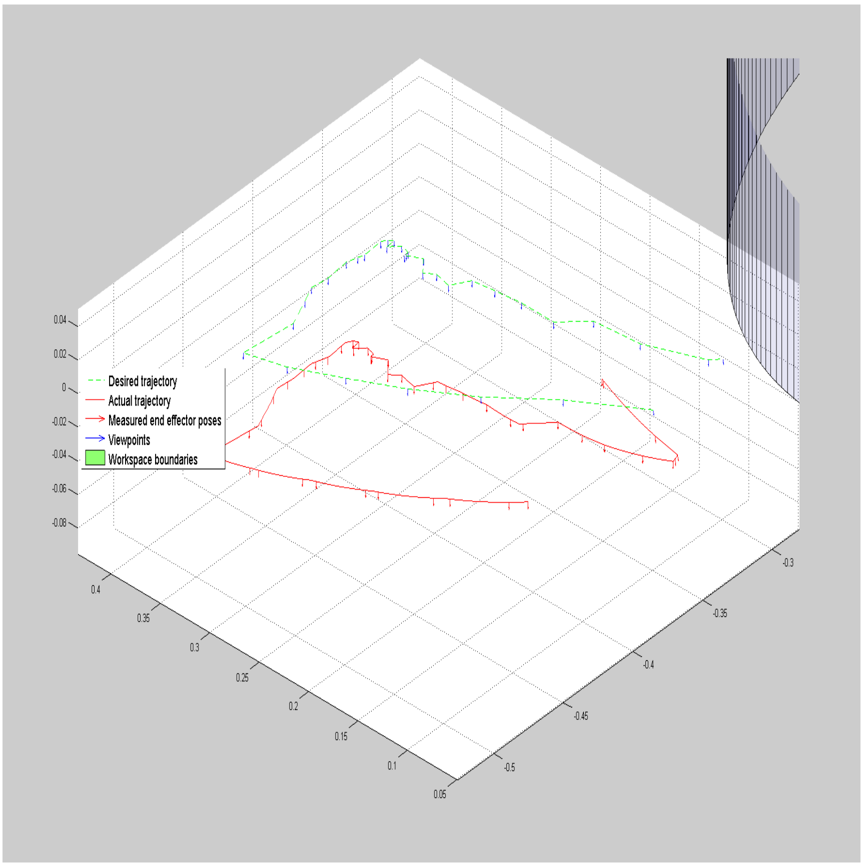
Robot trajectory cartesian motion planner mount

Figure 7.7 Results for the mount inspection plan eef_ step = 0.03 and jump treshold = 0.0



Robot trajectory IKMP mount

Figure 7.8 Results when simulating augmented algorithm generated path for the plane



Robot trajectory IKMP plane

Robot trajectory IKMP Cube

Figure 7.9 Results when simulating augmented algorithm generated path for the mount

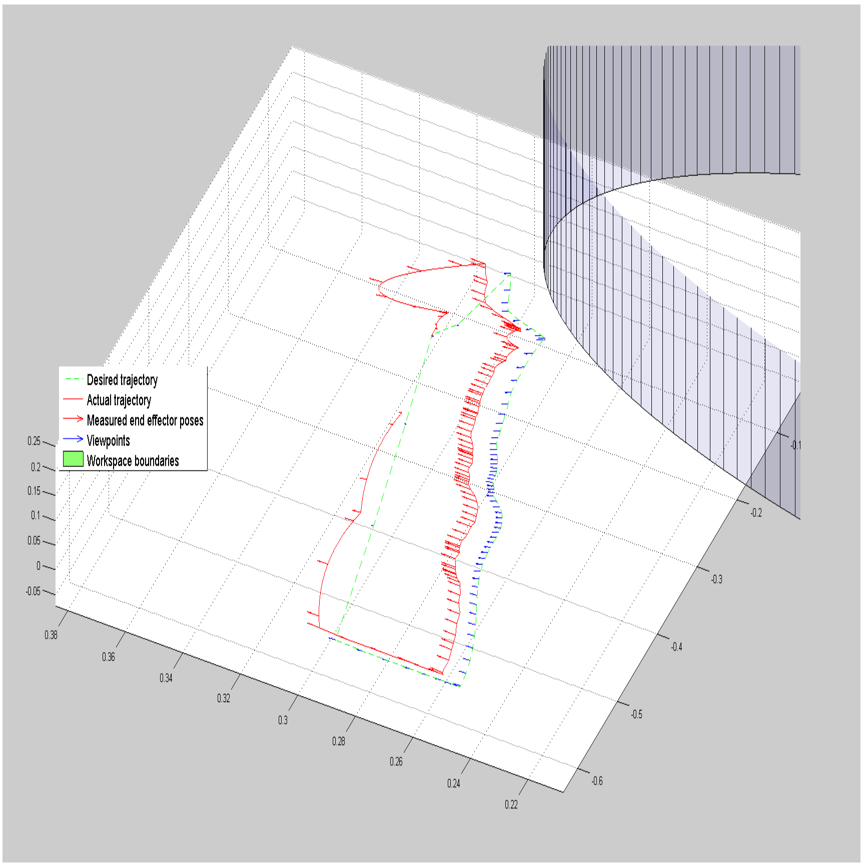
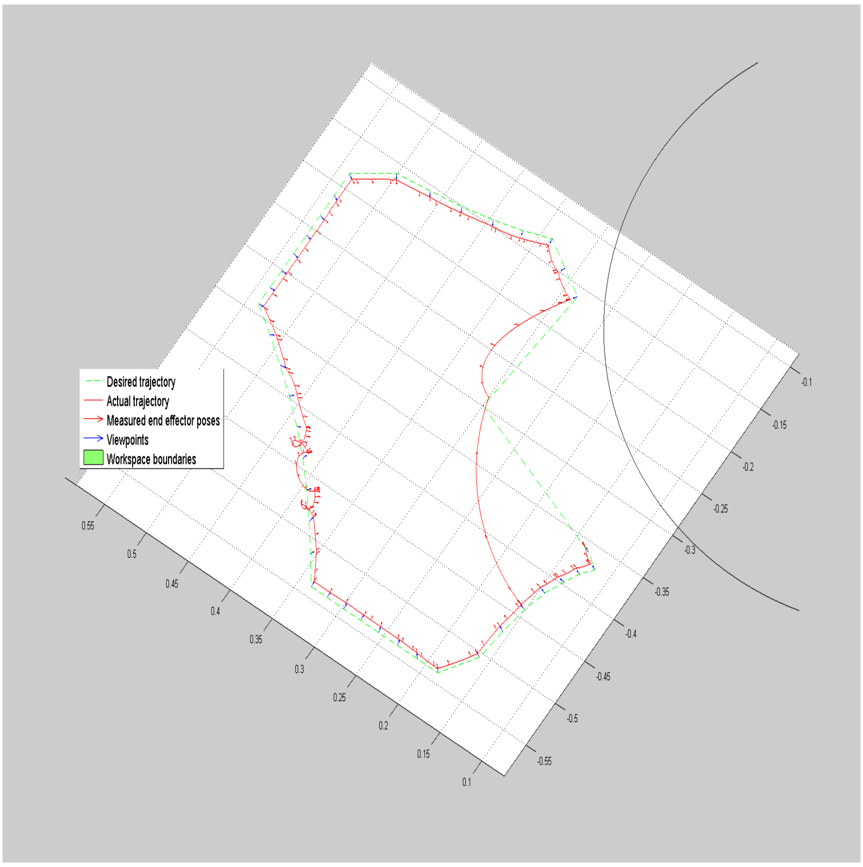


Figure 7.10 Results when simulating augmented algorithm generated path for the cube



Robot trajectory variations IKMP Cube

Figure 7.11 Results when simulating path resulting from inspection planning algorithm v0

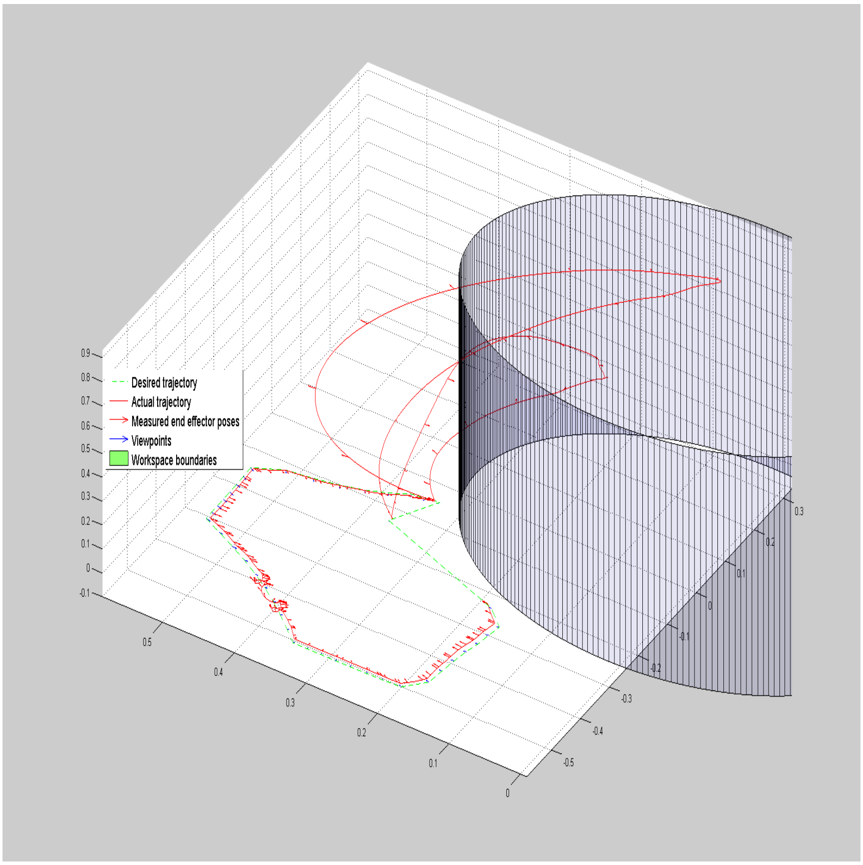


Figure 7.12 Results when simulating path from inspection planning algorithm v1

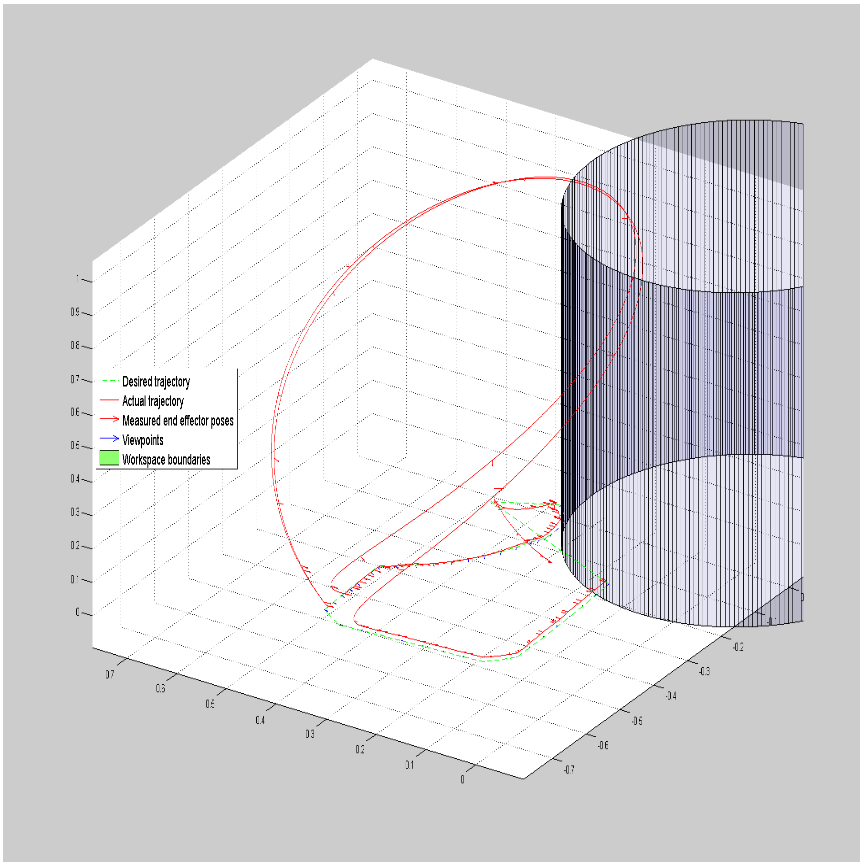


Figure 7.13 Results when simulating path resulting from inspection planning algorithm v2

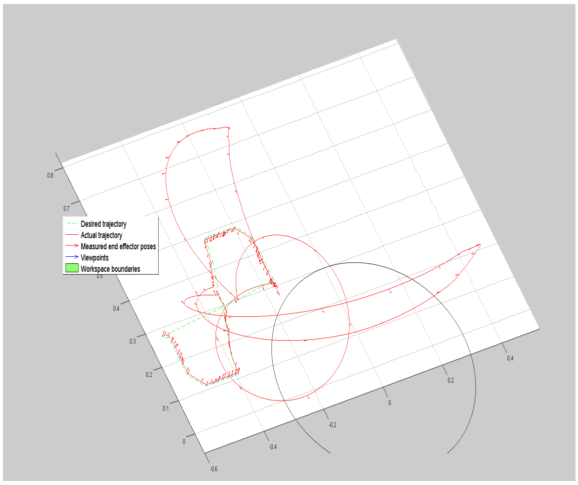


Figure 7.14 Results when simulating path resulting from inspection planning algorithm v3

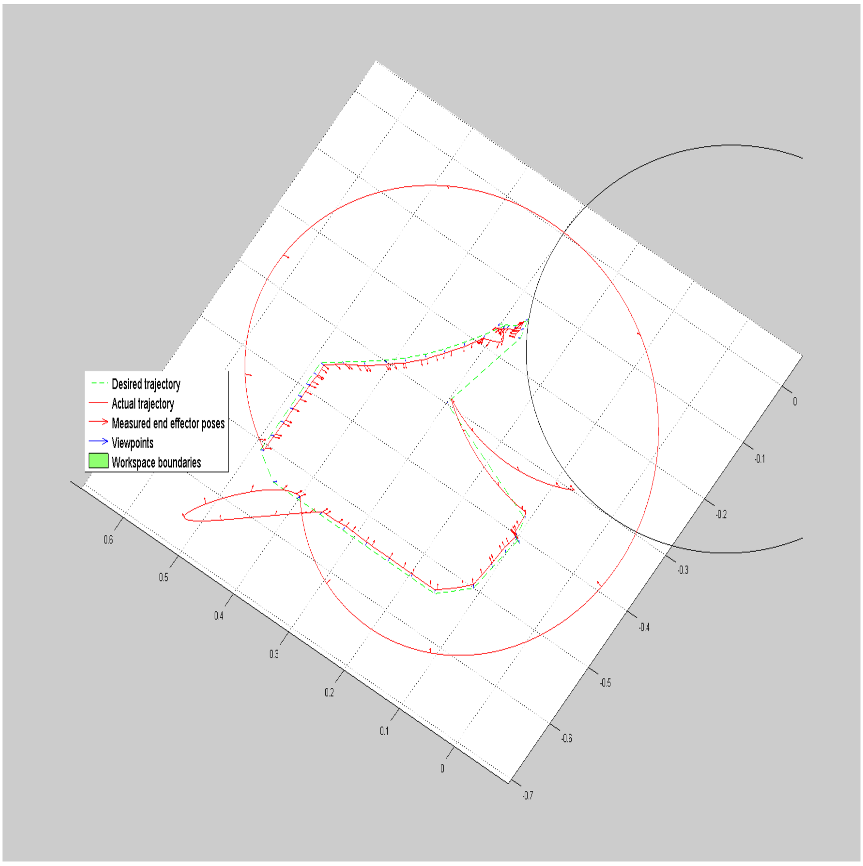
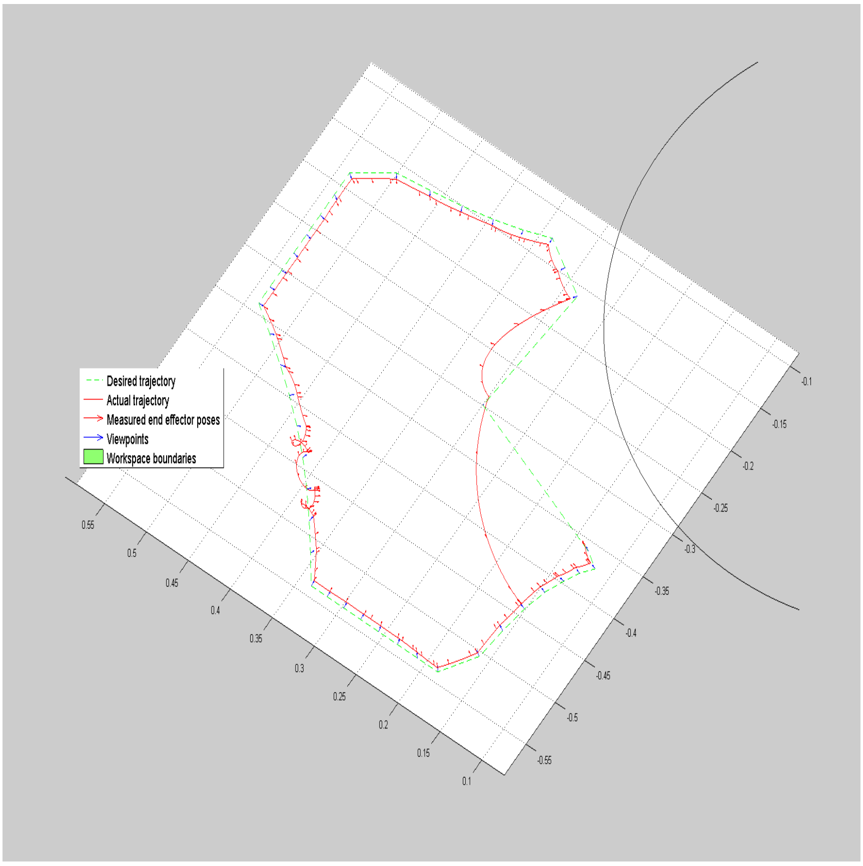


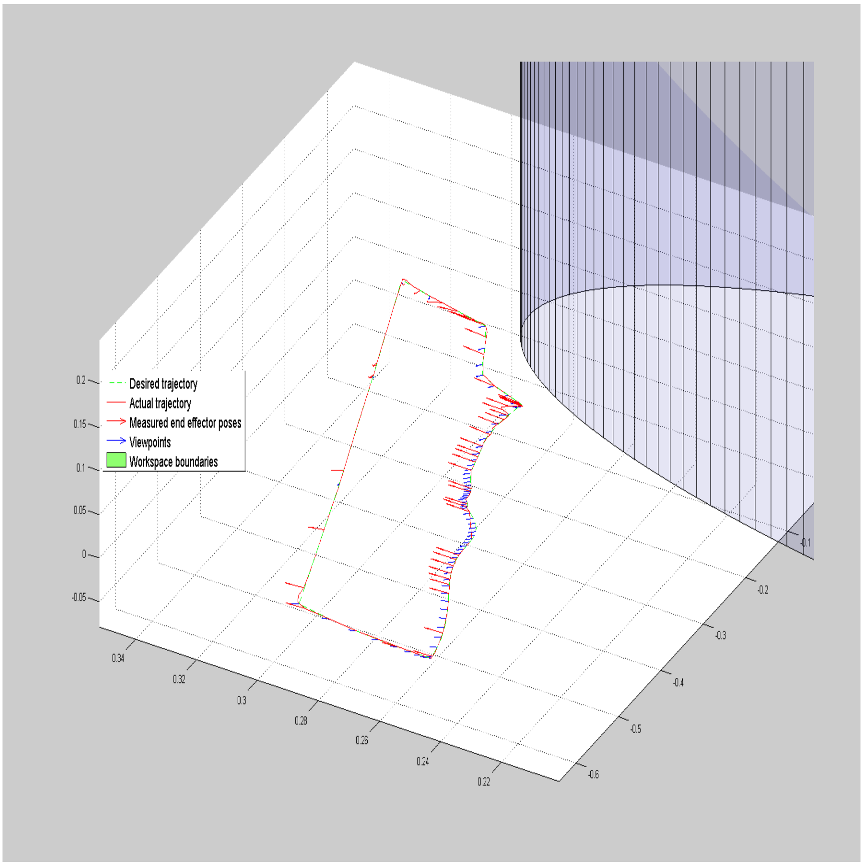
Figure 7.15 Results when simulating path resulting from inspection planning algorithm v4



7.6.3 Robot trajectories

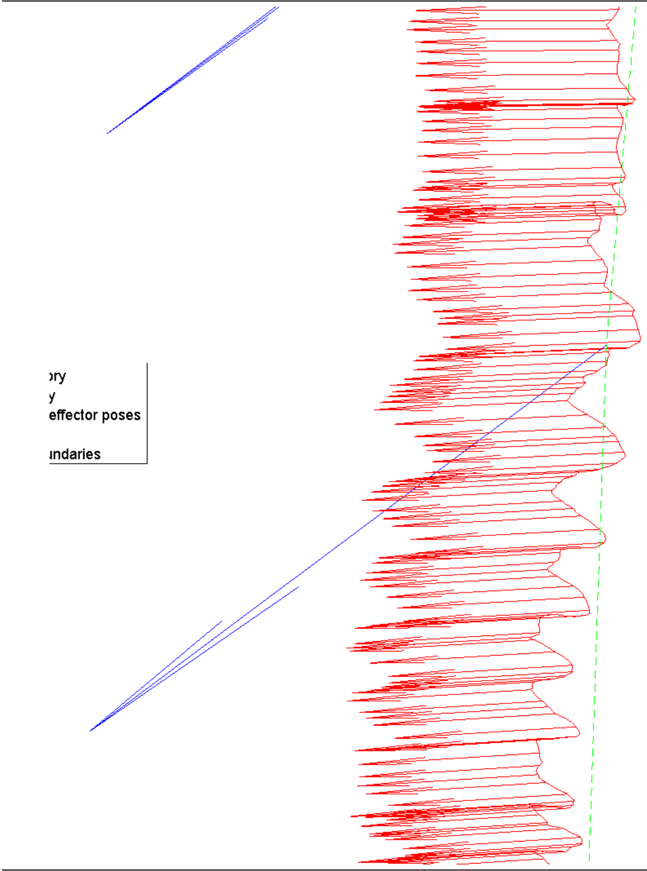
Robot trajectory, Cartesian planner, mount

Figure 7.16 Measured trajectory, camera mount inspection with cartesian motion planner



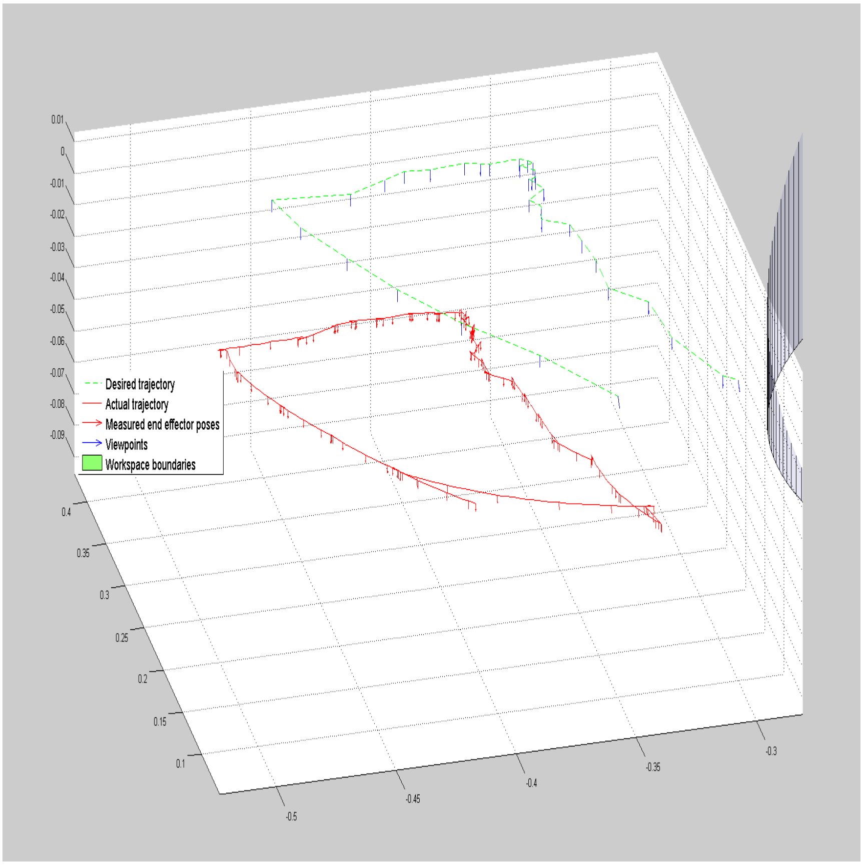
Cartesian planner orientation error

Figure 7.17 Visible fluctuations from the path as well as orientation error. Blue arrows are the specified orientation, measured orientation given by the red arrows



Robot trajectory, IKMP, plane

Figure 7.18 Measured trajectory, running the inverse kinematics based planner



Offset and pose error IKMP

Figure 7.19 Picture from above, showing overshoot from motion control. Overshoot looks like small branches from the main path

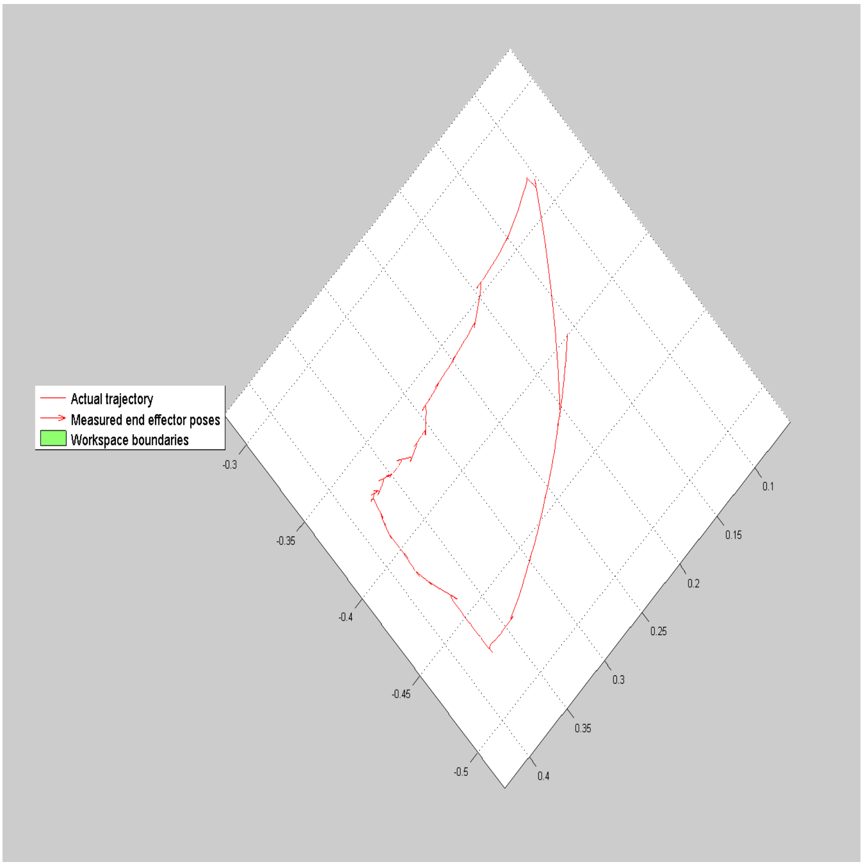
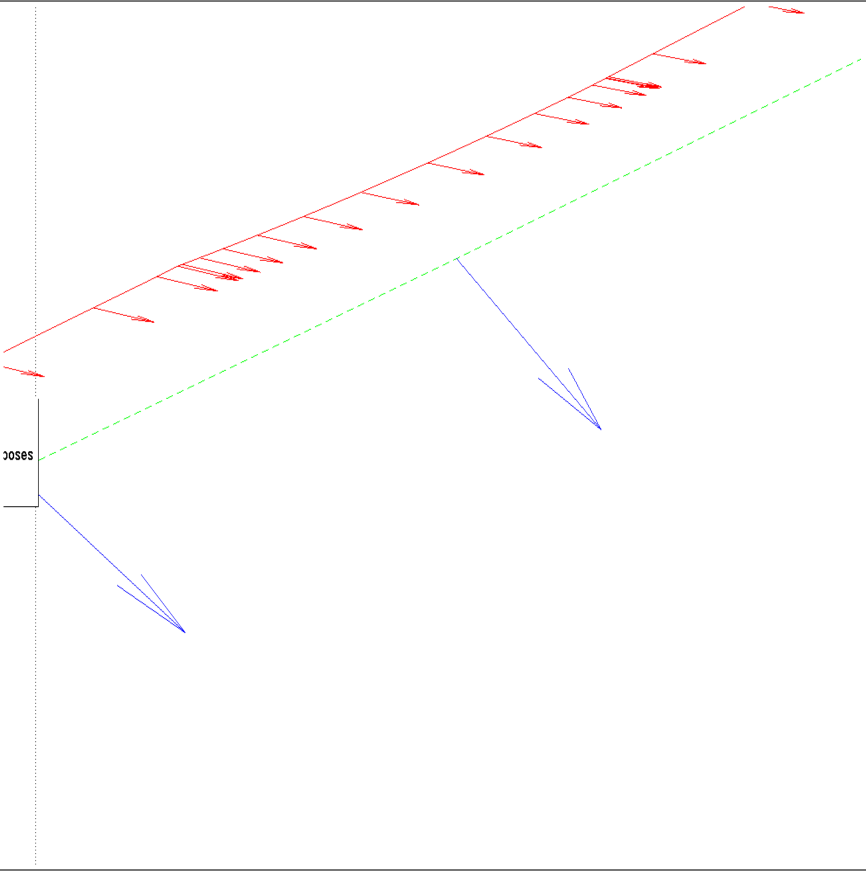


Figure 7.20 Error in achieving desired orientation. blue arrows are the specified orientation, measured orientation given by the red arrows



7.7 Code

All code related to simulations are available at `https://github.com/mmseines/ur5_simulation`.

7.7.1 Inspection planner modifications

Viewpoint feasibility criterion

Code is an outtake from Rotorcraft.hpp lines: 592 - 732

```
        /*
                  Ensure all selected viewpoints have valid IK solutions.
                */

                geometry_msgs::Pose viewpoint;
                tf::Quaternion qu = tf::createQuaternionFromRPY(0.0, g[4], g[5]);

                viewpoint.position.x = g[0]/100; //lage scale constant?
                viewpoint.position.y = g[1]/100;
                viewpoint.position.z = g[2]/100;
                viewpoint.orientation.x = qu.x();
                viewpoint.orientation.y = qu.y();
                viewpoint.orientation.z = qu.z();
                viewpoint.orientation.w = qu.w();
                bool found_ik = false; //  = g_robot_state.setFromIK(g_joint_model_group, viewpoint, 8, 0.1);

                collision_detection::CollisionResult c_res;
    for(int i = 0; i <3; i++){
      found_ik = g_robot_state.setFromIK(g_joint_model_group, viewpoint, 8, 0.1);
      if(found_ik == false){
        continue;
      }
      g_planning_scene->setCurrentState(g_robot_state);
      g_planning_scene->checkCollision(c_req, c_res);
      if(!c_res.collision){
        break;
      }
    }

    double c_pos = FLT_MAX * 0.9;
    double c_orient = FLT_MAX * 0.9;
                if(c_res.collision || !found_ik)
    {
                        std::vector<StateVector> solutions;
                        int iter = 0;
                        //atm only one solution...
                        while(solutions.size() < 5 && iter < 100)
                        {
                                StateVector tmp = g;
        double pos_displ[3];
        double ori_displ[3];
        for(int i = 0; i < 3; i++){
          pos_displ[i] = -0.9 + ((double)rand() / RAND_MAX) * (1.8);
          ori_displ[i] = -0.2 + ((double)rand() / RAND_MAX) * (0.4);
        }
        //Normalize both vectors.

                                tmp[0] = g[0] + pos_displ[0];
                                tmp[1] = g[1] + pos_displ[1];// -0.9 + ((double)rand() / RAND_MAX) * (1.8);
                                tmp[2] = g[2] + pos_displ[2]; //-0.9 + ((double)rand() / RAND_MAX) * (1.8);
                                tmp[4] = g[4] + ori_displ[1];
        tmp[5] = g[5]   + ori_displ[2];

                                if(this->isVisible(tmp) && !this->IsInCollision(tmp)) // this checks distance, indece angles and collision.
                                {
                                        qu = tf::createQuaternionFromRPY(0.0, tmp[4], tmp[5]);
```

```

viewpoint.position.
    x = tmp[0]/100;
    //lage scale
    constant?
viewpoint.position.
    y = tmp[1]/100;
viewpoint.position.
    z = tmp[2]/100;
viewpoint.
    orientation.x =
    qu.x();
viewpoint.
    orientation.y =
    qu.y();
viewpoint.
    orientation.z =
    qu.z();
viewpoint.
    orientation.w =
    qu.w();

found_ik =
    g_robot_state.
    setFromIK(
    g_joint_model_group
    , viewpoint, 8,
    0.1);
if(found_ik)
{
        g_planning_scene
            ->
            setCurrentState
            (
            g_robot_state
            );
        c_res.clear
            ();
        g_planning_scene
            ->
            checkSelfCollision
            (c_req,
            c_res);
        if(!c_res.
            collision
```

```
                                            )
    {
                                                solutions
                                                   .
                                                   push_back
                                                   (
                                                   tmp
                                                   )
                                                   ;
                                        }
                                }
                        }
                        iter++;
                }

                if(solutions.size()>0 && this->
                   initialized)
                {
int min_sol;
double cost_min = FLT_MAX * 0.9;
for(int ite = 0; ite < solutions.size(); ite++){
  //Orientation:
  StateVector tmp = solutions[ite];

  double dpsi1 = tmp[5]-alfa1;
  double dpsi2 = tmp[5]-alfa2;
  double dom1 = tmp[4] - omega1;
  double dom2 = tmp[4] - omega2;
  if(fabs(dpsi1)>M_PI)
    dpsi1 = 2*M_PI-fabs(dpsi1);
  if(fabs(dpsi2)>M_PI)
    dpsi2 = 2*M_PI-fabs(dpsi2);

  double orien_x = cos(tmp[4])*cos(tmp[5]);
  double orien_y = cos(tmp[4])*sin(tmp[5]);
  double orien_z = sin(tmp[4]);
  double dot_prod = ( orien_x*this->aabs[0] +
      orien_y*this->aabs[1] + orien_z*this->aabs[2]
      );

  c_orient = (1+dot_prod)*0.1 + sqrt(std::max(pow(
      dpsi1,2.0)/(dp),pow(dom1,2.0)/(dp)) + std::max
      (pow(dpsi2, 2.0)/(ds),pow(dom2,2.0)/(ds)));

      //Objective: ()
      c_pos = 0;
      for(int i = 0; i < 3; i++)
      {
        c_pos += pow(tmp[i] - (*state1)[i], 2.0);
        c_pos += pow(tmp[i] - (*state2)[i], 2.0);
        c_pos += g_const_D*pow(tmp[i] - (*statePrev)[i], 2.0);
      }

      if(c_pos+c_orient < cost_min)
      {
        min_sol = ite;
        cost_min = c_pos + c_orient;
      }

    }
    StateVector sol = solutions[min_sol];
    for(int i = 0; i <6; i++)
    {
      g[i] = sol[i];
    }
                    }
  else if(solutions.size() == 0 && this->initialized)
  {
                            solFoundLocal = false;
                    }
            }

    //Why run orientation optimization if solFoundLocal is set to false?.
if(c_pos + c_orient < cost && solFoundLocal){
  best = g;
  cost = c_pos + c_orient;
}
else if(this->VPSolver->getObjVal()+xxCompensate+costOrientation<cost && solFoundLocal)
{
  best = g;
  cost = this->VPSolver->getObjVal()+xxCompensate+costOrientation;
}
solFound |= solFoundLocal;
```

Lazy distance evaluation

Code snippet from PTPplanner.cpp in the inspection planner. Lines shown: 879 - 1069.

```
int cplusplus_callback_function(int ID, int ID2)
{
  ID%=maxID;
  ID2%=maxID;
#ifdef __TIMING_INFO__
  timeval time;
  gettimeofday(&time, NULL);
  time_LKH += time.tv_sec * 1000000 + time.tv_usec;
#endif
  long ret = INT_MAX;

  if(!plannerArrayBool) // allocate
  {
    plannerArray = new PTPPlanner*[maxID];
    for(int j = 0; j<maxID; j++)
    {
      plannerArray[j] = NULL;
    }
  }
  if(reinitRRTs[ID] == 1&&plannerArray[ID]) // delete for
     reinit
  {
    delete plannerArray[ID];
    plannerArray[ID] = NULL;
  }
  if(reinitRRTs[ID] == 2&&plannerArray[ID])
  {
    for(int j = 0; j<g_rrt_it_init; j++)
      plannerArray[ID]->rrts_.iteration();
    reinitRRTs[ID] = 0;
  }
  if(!plannerArrayBool) // build first tree, that is not
     built otherwise
  {
    plannerArrayBool = true;
    StateVector tmp = VP[0];
    plannerArray[0] = new PTPPlanner();
    plannerArray[0]->initialize(tmp,tmp[0],tmp[1],tmp
       [2],2.0*g_rrt_scope,2.0*g_rrt_scope,2.0*g_rrt_scope)
       ;
  }
  if(!plannerArray[ID]) // init
  {
    plannerArray[ID] = new PTPPlanner();
    StateVector tmp = VP[ID];
    plannerArray[ID]->stateVec = tmp;
    plannerArray[ID]->initialize(tmp,tmp[0],tmp[1],tmp
       [2],2.0*g_rrt_scope,2.0*g_rrt_scope,2.0*g_rrt_scope)
       ;
    reinitRRTs[ID] = 0;
  }


  double distLazy = sqrt( pow(VP[ID][0] - VP[ID2][0],2.0) +
      pow(VP[ID][1] - VP[ID2][1],2.0) + pow(VP[ID][2] - VP[
     ID2][2],2.0) );

  if(distLazy>g_rrt_scope)
  {
#ifdef __TIMING_INFO__
    gettimeofday(&time, NULL);
    time_LKH -= time.tv_sec * 1000000 + time.tv_usec;
#endif
    bool bCollision = false;

                  int numSelfCol = 0;
    // collision check also for lazy connections
    if(g_lazy_obstacle_check)
    {
                          //self collision check.
                          planning_scene::PlanningScenePtr
                             g_planning_scene(new
                             planning_scene::PlanningScene(
                             g_robot_mdl));

      moveit_msgs::PlanningScene planning_sc;
      //- -------------------------- adding obstacles.
         -------
      moveit_msgs::CollisionObject table;
      table.id = "table";
      shape_msgs::SolidPrimitive primitive;
      primitive.type = primitive.BOX;
      primitive.dimensions.resize(3);
      primitive.dimensions[0] = 1.0;
      primitive.dimensions[1] = 1.0;
      primitive.dimensions[2] = 0.2;

      table.primitives.push_back(primitive);

      table.header.frame_id = "world";

      geometry_msgs::Pose table_pose;
      table_pose.position.z = -0.33;
      table_pose.position.x = -0.2;
      table_pose.position.y = 0.0;
      table_pose.orientation.x = 0.0;
      table_pose.orientation.y = 0.0;
      table_pose.orientation.z = 0.0;
      table_pose.orientation.w = 0.0;

      table.primitive_poses.push_back(table_pose);

      table.operation = table.ADD;
      planning_sc.world.collision_objects.push_back(table);

      moveit_msgs::CollisionObject wall;
      wall.id = "wall";
      primitive.type = primitive.BOX;
      primitive.dimensions.resize(3);
      primitive.dimensions[0] = 0.1;
      primitive.dimensions[1] = 2.0;
      primitive.dimensions[2] = 1.5;

      wall.primitives.push_back(primitive);

      wall.header.frame_id = "world";

      geometry_msgs::Pose wall_pose;
      wall_pose.position.x = 0.0;
      wall_pose.position.y = -0.4;
      wall_pose.position.z = 0.0;

      tf::Quaternion q = tf::createQuaternionFromRPY(0.0,
          0.0, -M_PI/4.0);
      wall_pose.orientation.x = q.x();
      wall_pose.orientation.y = q.y();
      wall_pose.orientation.z = q.z();
      wall_pose.orientation.w = q.w();

      wall.primitive_poses.push_back(wall_pose);

      wall.operation = wall.ADD;
      planning_sc.world.collision_objects.push_back(wall);

      planning_sc.is_diff = true;
      g_planning_scene->setPlanningSceneDiffMsg(planning_sc);
      // ------------- end add collisions.

               robot_state::RobotStatePtr kinematic_state(new robot_state::RobotState(g_robot_mdl));
               robot_state::RobotState g_robot_state(g_robot_mdl);
               const robot_state::JointModelGroup * g_joint_model_group = g_robot_mdl->getJointModelGroup("manipulator");

               collision_detection::CollisionRequest c_req;
               c_req.contacts = 1;

      for(double it = 0; it < 1; it += g_discretization_step/distLazy)  // Inperpollation where colision is checked.
      {
       double * tmp = new double[DIMENSIONALITY];
       for(int i = 0; i<DIMENSIONALITY; i++)
        tmp[i] = VP[ID][i]*it+VP[ID2][i]*(1-it);

                   //set from IK, the interpolated state.
       if(plannerArray[ID]->rrts_.system->IsInCollision(tmp))
       {
        bCollision = true;
        break;
       }else{

        geometry_msgs::Pose viewpoint;
                       tf::Quaternion qu = tf::createQuaternionFromRPY(0.0, tmp[4], tmp[5]);

```

```
                                    viewpoint.position.x = tmp[0]/100; //lage scale constant?
                                    viewpoint.position.y = tmp[1]/100;
                                    viewpoint.position.z = tmp[2]/100;
                                    viewpoint.orientation.x = qu.x();
                                    viewpoint.orientation.y = qu.y();
                                    viewpoint.orientation.z = qu.z();
                                    viewpoint.orientation.w = qu.w();
                                    bool found_ik = g_robot_state.setFromIK(g_joint_model_group, viewpoint, 8, 0.1);
                                    g_planning_scene->setCurrentState(g_robot_state);

                                    collision_detection::CollisionResult c_res;
            if(!found_ik){
              numSelfCol++;
              continue;
            }
                                    g_planning_scene->checkSelfCollision(c_req, c_res);
                                    if(c_res.collision){
              numSelfCol++;
            }
```

```
                                }
        delete[] tmp;
      }
    }
    if(!bCollision)
    {
      return (int) std::min((double)INT_MAX-1,(((double)(distLazy * g_scale)) + 0.5 + 5.0*g_scale*numSelfCol));
    }
  }
  state_t state;
  state.setNumDimensions(DIMENSIONALITY);
  for(int j = 0; j<DIMENSIONALITY; j++)
    state[j] = VP[ID2][j];
  ret = (int)std::min((double)INT_MAX-1,(plannerArray[ID]->rrts_.evalDist(state)*g_scale+0.5)); // distance
#ifdef __TIMING_INFO__
  gettimeofday(&time, NULL);
  time_LKH -= time.tv_sec * 1000000 + time.tv_usec;
#endif
  return ret;
}
```

7.7.2 Misc.

This code runs even though the *USE_ FIXEDWING_ MODEL* constant is undefined.

Smoothing parameter

found at line 519 - 553 in plan.cpp from the inspection planner package.

```
#ifndef USE_FIXEDWING_MODEL

        int lim = 0;
        double smoothing_param = std::min(req.numIterations
            /2,25);
        double neighbour_steps = std::min(2.0, 1.0+50.0/req
            .numIterations);
        int maxWidth = (int)std::min(neighbour_steps*(
            double)(req.numIterations-smoothing_param-
            koptPlannerIteration), (double)(maxID)/2.0);

        while((*s1)[0] == plannerArray[Npred]->stateVec[0]
            && (*s1)[1] == plannerArray[Npred]->stateVec[1]
            && (*s1)[2] == plannerArray[Npred]->stateVec[2]
            && (lim++)<maxWidth)
        {
          if(kpred == 0)
            kpred = maxID-1;
          else
            kpred--;
          int NpredOld = Npred;
          Npred = PTPPlanner::Tour_[kpred]-1;
          *s1 = plannerArray[Npred]->rrts_.getLastVisible(
              plannerArray[NpredOld]->stateVec, tri[i]);
        }

#endif
        *s2 = plannerArray[Nsucc]->rrts_.getLastVisible(
            plannerArray[i]->stateVec, tri[i]);
#ifndef USE_FIXEDWING_MODEL

        lim = 0;
        while((*s2)[0] == plannerArray[Nsucc]->stateVec[0]
            && (*s2)[1] == plannerArray[Nsucc]->stateVec[1]
            && (*s2)[2] == plannerArray[Nsucc]->stateVec[2]
            && (lim++)<maxWidth)
        {
          if(ksucc == maxID-1)
            ksucc = 0;
          else
            ksucc++;
          int NsuccOld = Nsucc;
          Nsucc = PTPPlanner::Tour_[ksucc]-1;
          *s2 = plannerArray[Nsucc]->rrts_.getLastVisible(
             plannerArray[NsuccOld]->stateVec, tri[i]);
        }

#endif
```