UNIVERSITÄT BREMEN

MASTER THESIS

Tactile-Based and Cognition-Enabled Manipulation for Real World Assembly Tasks

Author:
Arthur NIEDZWIECKI

Supervisor:
Prof. Dr. h.c. Michael BEETZ, PhD
Second Supervisor:
Dr. rer. nat. Federico RUIZ Ugalde
Advisor:
Gayane KAZHOYAN

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Institute for Artificial Intelligence
Computer Science

November 11, 2020

Declaration of Authorship

I, Arthur NIEDZWIECKI, declare that this thesis titled, "Tactile-Based and Cognition-Enabled Manipulation for Real World Assembly Tasks" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: [illegible]

Date: Bremen, den 12.11.2020

UNIVERSITÄT BREMEN

Abstract

Faculty 3
Computer Science

Master of Science

Tactile-Based and Cognition-Enabled Manipulation for Real World Assembly Tasks

by Arthur NIEDZWIECKI

Robotic arms are an important asset in efficient automation industry. Their inexhaustible capabilities for repetitive tasks outperform those of human beings by far. In the past years also the demand for varying assembly tasks has risen. Adapting an industrial robot to novel environments usually requires to manually adjust the robot's movement, even if the novelty only includes small changes in an object's position. Contemporary solutions involve autonomous robots, which are equipped with Artificial Intelligence to reason upon changes in the environment. Tasks designed for such robots need to be only general descriptions, whereby the robot interprets its procedure by actively perceiving its surroundings, enabling reaction to, and recovery from undesired situations. Using robot perception, however, produces inaccuracy in localization of a robot and objects equally, which makes intricate assembly tasks hard to perform.
This thesis aims to increase accuracy of an autonomous mobile robot during assembly tasks in the real word. Two approaches are designed for this, which utilize detection and evaluation of rigid-rigid contact forces between the gripper and an object. One approach updates an object's estimated position with respect to the robot by touching it, which inherently increases the robot's accuracy when manipulating it afterwards. The other one is applicable during immediate assembly of two parts, where the gripper adjusts its position incrementally over several trials by evaluating contact forces. The approaches are evaluated by assembling a Battat toy airplane in the real world. Said airplane is included in the Yale-CMU-Berkeley object dataset for manipulation benchmarks and consists of several irregular-shaped parts. Therefore it provides a contemporary and comparable setup for object manipulation in every-day environments.

UNIVERSITÄT BREMEN

Zusammenfassung

Fachbereich 3
Informatik

Master of Science

Tast- und Kognitions-basierte Manipulation für Zusammenbau-Aufgaben in der echten Welt

von Arthur NIEDZWIECKI

Roboterarme sind eine wichtige Investition in effizienter Automatisierungsindustrie. Ihre unerschöpflichen Möglichkeiten in sich wiederholenden Aufgaben übertreffen jene von Menschen bei Weitem. In den vergangenen Jahren ist auch der Bedarf nach veränderlichen Montage-Aufgaben gestiegen. Um einen industriellen Roboter an eine neue Umgebung anzupassen, müssen seine Bewegungen üblicherweise manuell eingestellt werden, auch wenn die Neuerung lediglich kleine Veränderung in der Position von Objekten beinhaltet. Zeitgenössische Lösungen involvieren autonome Roboter, welche, mit Künstlicher Intelligenz ausgerüstet, über die Veränderung ihrer Umgebung nachdenken können. Aufgaben für solche Roboter brauchen nur abstrakte Beschreibungen zu sein, wonach der Roboter, durch Wahr- nehmung seiner Umgebung, seine Prozedur selbst interpretiert, und er auf ungewollte Situationen reagieren, und sich daraus befreien kann. Leider kommen durch robotische Wahrnehmung auch Ungenauigkeit einher, woduch komplizierte Aufgaben schwer zu bewältigen sind.
Diese Thesis zielt darauf ab die Genauigkeit von autonomen, mobilen Robotern bei Montage-Aufgaben in der echten Welt zu verbessern. Dafür wurden zwei Ansätze entwickelt, welche den Krafteinfluss zwischen einem Greifer und Objekt wahrnehmen und evaluieren. Einer der Ansätze aktualisiert die vermutete Position eines Objektes im Verhältnis zum Roboter durch Berührung, was wiederum die Genauigkeit des Roboter verbessert, wenn das Objekt im nachhinein von ihm manipuliert wird. Der andere Ansatz ist direkt im Zusammenbau zweier Objekte anwendbar, wo der Greifer seine Position inkrementell über mehrere Versuche, durch Evaluation der Kontaktkräfte, anpasst. Die Ansätze werden evaluiert, indem der Roboter ein Battat Spiel-Flugzeug in der echten Welt zusammenbaut. Das verwendete Flugzeug ist Teil des Yale-CMU-Berkeley Object Dataset, welches für Benchmarks in Robotermanipulation verwendet wird, und besteht aus mehreren unregelmäßig geformten Bauteilen. Deshalb bietet es ein zeitgenössisches und vergleichbares Beispiel für die Manipulation von Objekten in alltäglichen Umgebungen.

Contents

Chapter 1

Introduction

1.1 General Approach and Research Questions

Industrial assembly nowadays is supported by a variety of machines and robots, performing the same task over and over again to account for tasks usually dangerous or exhausting for a human being. In recent years the demand for industrial robots has risen and their spectrum of applicability became incrementally richer. For each task a new robot is designed and programmed to solve it. Some robots can be re-used from one task to another by changing their movement, adjusting it to novel environments. Contemporary research already tries to design industrial robots that collaborate with humans for a common goal, where the robot reacts to human activities and changes its movement accordingly (Sun et al., 2020). But on their own, robots rely heavily on detailed knowledge about their immediate environment to autonomously make decisions on what to do next, and how to do it. Depending on the accuracy of such knowledge, intelligently planned movement can be very precise, or completely fail their goal. Autonomous robots have been, and still are heavily researched. Industrial arms achieve their precision by being manually programmed to do the exact same thing. Especially in assembly tasks, where high accuracy is imperative, robots without any knowledge of what they are doing can be extremely precise in one task, but it takes only one misplaced part to render such robots incapable. On the other hand, autonomous cognitive robots are supposed to be general, reacting to novel situations by adjusting themselves. This requires perception of the environment, which is never completely flawless.

With the help of a cognitive system a robot is able to keep track of its surroundings and the changes within, including the parts it is tasked to assemble together. This world-state representation helps robots to find their way around, navigating while avoiding collisions, locating objects etc., but since the accuracy of an autonomous robot in a real-world environment depends heavily on its sensors, an object's correct location can only be estimated to a certain degree. Combining the abilities of a cognitive system with sensor measurements in the correct places, the accuracy can be greatly improved. Localizing the robot is typically done with laser sensors and odometry measurements, retrieved from optical encoders in the wheel motors of the mobile base. Object locations are estimated by applying computer vision algorithms to data coming from RGB-D sensors. Accuracy in general relies on the accuracy of the sensors and algorithms to evaluate their emitted data.

To improve accuracy, one can use force-torque sensors to gain tactile information. Robot's arms have optical encoders for each joint, enhanced with torque sensors to improve their estimated state, which is used to know where the arm is in space. Force-torque sensors can be equipped to the arm's end effector to receive contact forces. Knowing where the arm is, combined with contact detection between the

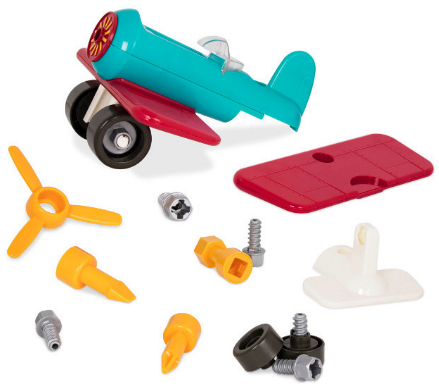


FIGURE 1.1: Battat airplane parts for assembly

arm and the environment, can be used to improve the estimates of object locations with respect to the robot.

In this thesis I want to show how the assembly process, executed by an autonomous robots in a real-world environment, high-level controlled by a cognitive system using a gaming engine as knowledge representation, can benefit from utilizing force-torque sensors to increase accuracy and feasibility of such intricate manipulation tasks. In two different approaches I will discuss how force response can be helpful, and where its limits might be.

1.2 Contributions

Assembly tasks are usually designed for robotic arms in precise environments. My work shows, that the combination of a cognitive environment, gaming-engine-based spatial reasoning, reactive planning and simple evaluations of force feedback can improve the accuracy of autonomous, humanoid robots in uncertain environments. Two approaches improve the assembly of a toy airplane (see figure 1.1), executed by an autonomous robot. This airplane is included in the Yale-CMU-Berkeley dataset, which is used for manipulation benchmarks.

In the first approach, contact forces are used to adjust the gripper's pose during assembly. To assemble two parts of the airplane, one part is held by the gripper and placed onto another part. If the actuated part is not correctly aligned, rotational velocity is received in the force-torque sensor. Depending on the direction of misalignment different torques emerge, which are reasoned upon to adjust the gripper until the assembly is successful.

A second approach aims to identify the actual pose of an object with respect to the robot before manipulating it. By touching the object in the real world the estimate of its pose can be adjusted. To calculate the adjustment a gaming engine is used to estimate an expected contact point between gripper and object, which is compared to the actual contact obtained by the gripper's position in the real world, depending on the arm's configuration during contact. Improving an object's believed pose inherently improves the robot's accuracy when manipulating it.

Both procedures are tested with every-day shaped objects, namely the airplane parts. The first approach is evaluated quantitatively in simulation and both approaches on the real-world robot.

1.3 Related Work

The following section puts this thesis into perspective with related and similar contemporary work. Two different approaches are formulated to assemble a toy airplane in an uncertain environment. The Yale-CMU-Berkeley Object and Model Set (Calli et al., 2015) where this airplane is included, has been recently considered in (Yang et al., 2020) for benchmarking their gripping capability. Therefore, assembling this airplane designs a suitable and contemporary task.

Each airplane part is of different shape. Evaluating contact forces between non-primitive shapes is a complex problem, because contact forces between these shapes are highly irregular. Research on standard Peg-In-Hole tasks (Wong, 1975), classifying contact forces to adjust gripper positioning for successful execution through deep learning is not applicable, because the observed experiments usually include primitive shapes of cylinders and regular cubes. A much simpler, more generic heuristic classifier, based on the observations in (Bouchard et al., 2015), is designed to work in collaboration with the underlying cognitive framework CRAM (Mösenlechner, 2016). Similar to the the work in (Muxfeldt and Kubus, 2016), classification of the current state of assembly is done by transferring sub-symbolic into symbolic representation, in order to reason upon and react to collisions in a comprehensible way.

Assembling the airplane parts can also be described as insertion tasks, since each part needs to fit properly on another. Recent work on industrial insertion tasks makes use of various sensors. In (Schoettler et al., 2019) for example, a reinforcement-learning based approach is designed, which makes use of image processing to specify rewards. Later, an other approach was release by the same author (Schoettler et al., 2020). Compared to his previous work, the presented reinforcement-learning technique was adapted to learn insertion in simulation, to decrease the amount of trials in real-world. Similar to his work, this thesis tries to successfully execute insertion tasks under uncertainty, however, the correction of error here is obtained through contact forces, and the underlying mechanisms are embedded within a cognitive framework (CRAM) using symbolic and sub-symbolic reasoning.

Tactile perception has recently been used to detect contact events through wiping actions (Stelter, Bartels, and Beetz, 2018). Stelter's work was considered in this thesis for classification of the gripper's positioning error during assembly, but the diversity of shapes would imply major changes to Stelter's existing system and requires reliable data for training, which is difficult to acquire in my experiments. Classification of shapes and material through tactile arrays, as performed by (Taddeucci et al., 1997) and (Luo et al., 2017), is not necessary, since all shapes are known a prior. Instead, the general idea of (Ruiz-Ugalde, Cheng, and Beetz, 2011) is reflected in this thesis, where the original position of an object is known through image processing, but updating an object's position is done with contact forces. A CRAM specific concept of continuous reaction to conditions is combined with the capabilities of gaming engines to find the position of objects.

Incorporating gaming engines into reasoning frameworks is followed in contemporary research, as shown in (Haidu et al., 2018), where the Unreal Engine is used for bleeding edge reasoning tasks in everyday environment. In this thesis the Bullet Physics Library (Coumans, 2015) is used to simulate the robot's surroundings, which enables spacial reasoning on the airplane's parts. It provides collision detection and contact point estimation between simulated objects.

1.4 Reader’s Guide

Over four chapters the contribution of this thesis is presented.

Chapter 2 Foundations explains the existing components on which this research is based, including the cognitive machine CRAM, the physics environments Gazebo and Bullet, the RViz visualization tool, constraint- and optimization-based controller, called Giskard and a description of Boxy, the robot used for real-world application.

Chapter 3 Methods and Implementation describes the two different approaches that I developed to integrate force response in cognitive procedures for improving robot performance in assembly tasks.

Chapter 4 Experimental Evaluation puts both approaches to the test, in simulation as well as in the real world on Boxy, where a toy airplane is to be assembled. Also several difficulties in working with real robots are elaborated.

Chapter 5 Conclusion summarizes the presented approaches and their results, discussing the problems dealt with from a wider angle to bring it into perspective for future work.

Chapter 2

Foundations

All the software stacks and prior implementations used throughout the thesis are explained in this chapter. It contains a general overview of the Robot Operating System (ROS) and a description of Boxy, an autonomous robot which is used throughout all experiments. Controlling Boxy is administered by Giskard, whose involvement is briefly addressed and oriented on the principle of the Naive Kinematics Simulator for robots in simulation, explanation included. Afterwards the main belief state, Bullet Physics Library is shown and how its engine is utilized. Then a visualization through RViz is given, both helpful in simulation and real-world. Also used in this thesis is the Gazebo physics simulator, whose involvement is explained thereafter. Then a thorough explanation is given on the Cognitive Robot Abstract Machine (CRAM), used for designing the procedures described in chapter 3.

2.1 ROS

The Robot Operating System (Quigley et al., 2009) is a standard middleware and dynamic framework for communication between components of a larger control system to implement robust robot programs. It is mostly used within research projects in robotics and pitching demonstrations in industrial use. Multiple exchangeable nodes can be designed separately to solve one specific task, be it image processing, localization, joint control, high-level planning, etc., which in collaboration can achieve a greater goal. In this thesis it is used as middleware for communicating the robot state, occupancy map of the environment, object positions, Giskard's administration of all joint controllers, transmission of laser sensor data for localization and force-torque feedback, as well as high-level control and monitoring of all collaborating nodes, while everything above is visualized in RViz.

2.2 Boxy

Boxy is a mobile manipulation robot platform. In contrast to the PR2 from Willow Garage (Cousins, 2010; Bohren et al., 2011) or Toyota's HSR (Yamamoto et al., 2018), Boxy (see figure 2.1) is assembled by the staff from the Institute for Artificial Intelligence (IAI). Boxy evolved from TUM-Rosie, first described in (Maldonado, Klank, and Beetz, 2010), assembled in Munich and later built from scratch in Bremen. There are 4 omnidirectional Mecanum wheels driving its base, allowing for holonomic movement. With two LIDAR sensors (Light Detection and Ranging) attached at a front-left and back-right, which provides 360 degree data laser scanner data for localization with the base wheel's odometry, which is estimated by an Adaptive Monte Carlo Localization (AMCL) algorithm. A map of the environment was already present, so no mapping (e.g. through SLAM) was necessary.

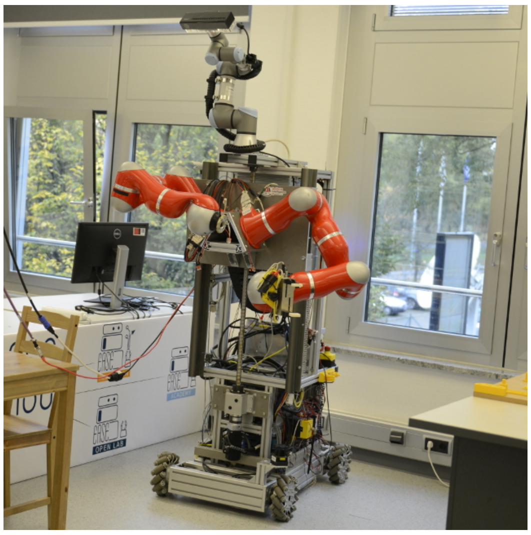


FIGURE 2.1: Humanoid robot called Boxy, used for manipulation tasks.

On top of Boxy's corpus a UR3 arm from Universal Robots is mounted with an X-Box One Kinect camera for processing 2D images and RGB-D data. During the implementation of this thesis' approach no image processing is performed, though. A torso can be moved prismatically, allowing up and down movement. Two arms are attached to Boxy's torso, both LWR-4+ 7-DOF lightweight arms from KUKA (Hirzinger et al., 2002; Burger et al., 2010). Mounted to the end effector is a six-dimensional Weiss KMS40 force-torque sensor (further called wrench sensor) and a Weiss WSG50 gripper with rubber fingertips to reduce slip. Both, the wrench sensor and gripper are connected via USB 3.0 to the main computer. Base motors communicate over Ethercat, and the LIDAR are connected via Fast Ethernet, while the arms are controlled over an independent Ethernet connection on a SERCOS bus to a designated computer. In combination with KUKAs LWR-4+ fiber-optic data transfer between the joints it allows communication of joint-velocities at up to 500Hz.

There are two on-board computers. One controls the LWR-4+ arms on a Debian Machine with real-time kernel. It provides a joint-impedance controller to the ROS network which makes it possible to move the arms from a separate machine, and provides the Links-And-Nodes middleware to manually control an arm's joints. The other computer serves as Ethercat Master to all sensors and base motors. It runs an Ubuntu 18.04 system with a low-latency kernel, using 32 CPU threads and 64 GB of RAM and is connected via fast Ethernet to its slaves in the base. On this machine, called Leela, runs the ROS core and all robot-dependent nodes like robot state broadcast, Emergency-Stop interruption, localization, teleoperation interface, torso, base and neck control, and navigation.

In the left image of figure 2.2 the wrench sensor's six axes are illustrated. On the right its orientation as end effector is shown. The gripper's and sensor's frame have the same orientation in the world. Later in this thesis the gripper's orientation changes throughout different assembly tasks, and with it the expected contact forces when the gripper is in collision with something in the real world.

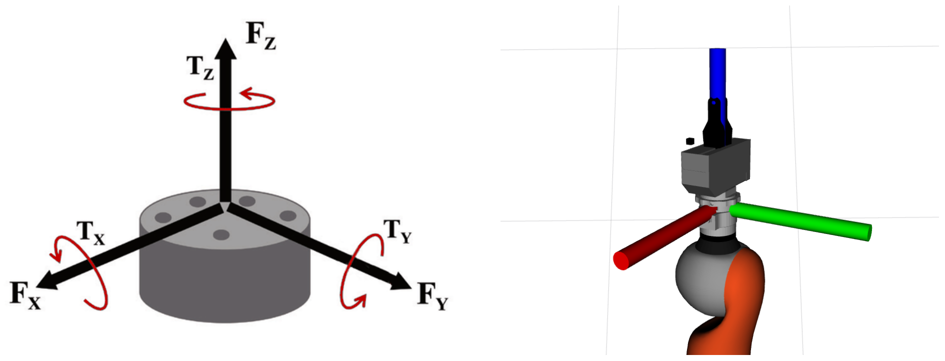


FIGURE 2.2: Left: Axes of a force-torque sensor (image taken from (Lee et al., 2018)), with axes adjusted to the particular sensor on Boxy. Right: Orientation of the Weiss KMS40 wrench sensor and WSG50 gripper on the LWR-4+ wrist.

2.3 Naive Kinematics Simulator

For comfortable simulation of a robot, the Naive Kinematics Simulator was developed in the IAI. This software is solely used during simulation of robot movement, not in real-world experiments. Its purpose is to have a simple kinematic representation of the robot, operating in ROS. It provides an interface to move not only the prismatic torso, UR3 neck and LWR-4+ arms, but also the robot's base as joint-velocity controllers. By this principle every aspect of a robot can be moved in the same way, and determining the robot's current status is done only through joint states, while every separate controller accounts for a subset of these joints.

2.4 Giskard

The constraint and optimization-based motion planner Giskard[1] can be used for moving a robot. Giskard can consider all joints, from base, torso and arms, into its trajectory calculation to achieve a certain goal represented as cartesian pose or joint state, e.g. when the gripper is to be put at a distant pose, all the usually disjoint controllers can collaborate to move the gripper into the desired position. Figure 2.3 illustrates this process for the PR2 robot, but is applicable for Boxy as well. On a given goal, Giskard calculates a sequence of joint-velocities for each involved joint, then splits the whole trajectory among the designated controllers, which execute their own joint-trajectory in parallel. For such a system to work, each controller must be designed to expect a trajectory of joint-velocities as input argument, which is an array of velocities per joint over a series of time. Such a controller moves its designated joints per time-step at the given velocity. Before this thesis, Boxy's controllers were addressed separately, without the described administration and motion control by Giskard. For the already available joint controllers on Boxy, new nodes were implemented as intermediate translators between Giskard and these low-level controllers.

To calculate trajectories for moving the gripper without causing collision, Giskard has an internal representation, called collision scene, of the robot's physique. Through

[1] `http://giskard.de/`

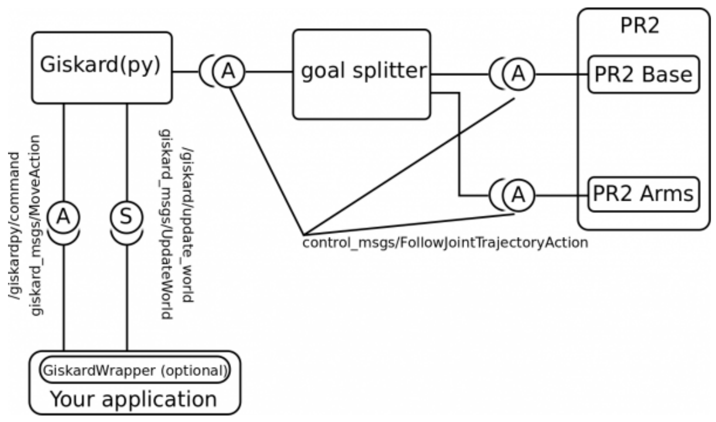


FIGURE 2.3: Image courtesy by Georg Bartels et. al. Giskard's procedure for administration of several joint-velocity controllers for the PR2 robot. Upon an incoming request from 'Your application' (bottom-left to top-left) a suitable trajectory for all joints is calculated (top-left) to achieve the requested goal. This trajectory is split per controller (middle) and send as commands (right) to each joint-controller. Interfaces (A) describe ROS actions, (S) services.

its URDF description (Unified Robotic Description Format) and all joint states, provided by each optical encoder of a joint, the current configuration of a robot is continuously up-to-date. Furthermore, the collision scene knows of the dimensions of all objects in the environment (see figure 2.4). An interface allows to update environmental changes as well, e.g. when the position of an object changes. Moving an object through manipulation with the gripper is automatically accounted for as well: by attaching a grasped objects to the gripper's link in the collision scene, the object moves together with the specified link. This allows for collision avoidance even between a held object and another. Giskard's internal collision scene is synchronized with the belief state, a simulated representation of the robot's surroundings in the gaming engine Bullet Physics, which is elaborated in section 2.5.

During certain trajectories, some joints can be restricted from Giskard's calculations. This functionality is used during precise movements throughout this thesis, especially the base is kept still in some sequences. For that, the interface between our high level control CRAM and Giskard was extended to allow such constraints.

2.5 Bullet Physics

The Bullet Physics Library (further abbreviated as Bullet) is a physics engine, designed for the simulation of collision detection, soft and rigid body dynamics (Coumans, 2015). Bullet provides tools for spatial reasoning and simulation of rigid bodies. In the top-middle image of figure 2.4 the Bullet world is shown with Boxy in front of the assembly board with all airplane parts. The shapes look relatively rough, because it only uses collision shapes instead of visuals from any CAD model.

Each significant object in the environment has its representation in Bullet and collision between then can be properly calculated. Bullet's collision detection alone has a huge benefit when simulating robot movement in cluttered environments, to find suitable positions and trajectories without damaging anything in the real world. In the current setup, however, Giskard takes care of collision avoidance, while Bullet

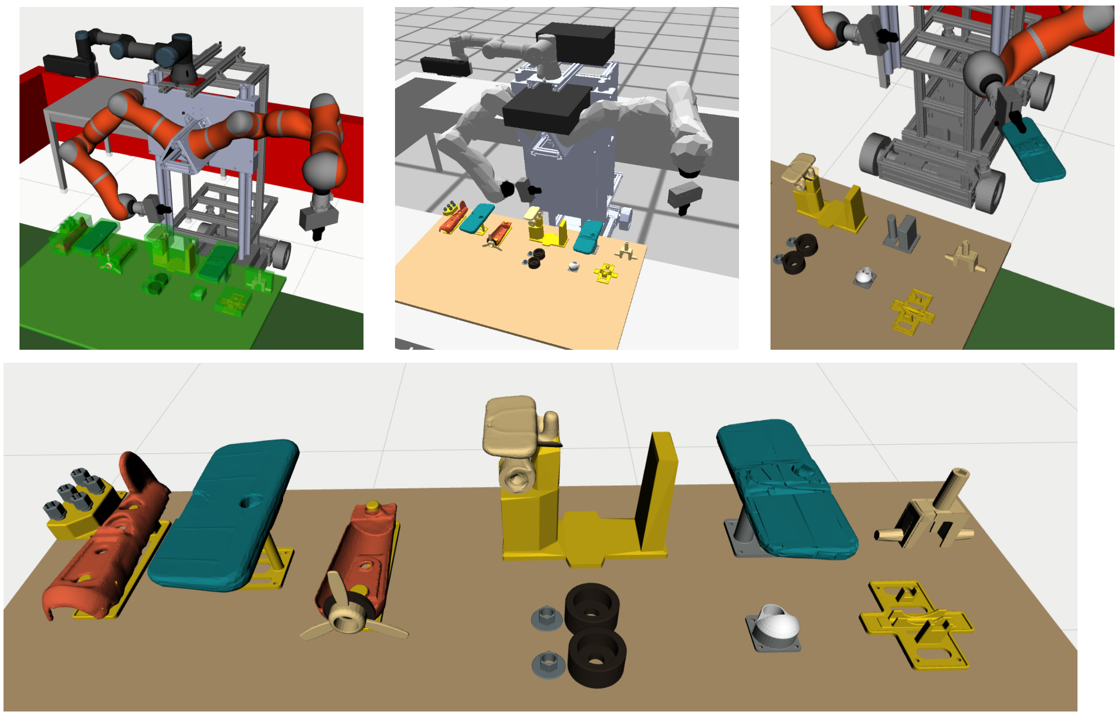


FIGURE 2.4: Assembly scene with Boxy and all airplane parts on a board. (top-left) RViz with Giskard collision shapes, (top-middle) Bullet Physics. The black boxes serve as spatial constraints for collision avoidance when moving the head. (top-right) Moving *bottom-wing* object part with the gripper. (bottom) Battat airplane parts, included in the YCB dataset.

is mainly used to update Giskard's collision scene, for object-related reasoning and determining contact points between rigid bodies.

Since the robot state is published over the whole ROS network, Bullet can adjusts its internal knowledge on Boxy's configuration as well. Instead of a continuous synchronization, as in Giskard, Bullet updates its robot state only after every terminated movement. This is due to Bullet's original purpose for our high-level control CRAM, where the simulated robot can rapidly teleport between states to immediately check if certain configuration are causing collisions. Also Bullet's robot state can be updated imperatively, e.g. after the robot was moved through teleoperation. Because Giskard takes care of motion planning, this restriction doesn't influence performance or accuracy, as long as both Bullet and Giskard are kept in sync.

Simulation of object manipulation is done in Bullet as in Giskard, where a simulated shape is attached to the gripper's as part of the robot's URDF, moving both simultaneously as can be seen in the top-right image in figure 2.4. With CRAM in Bullet, moving an object in the gripper is not continuous like in Giskard, but discretely done by teleporting an object along the same transformation, as its origin. But not only the gripper can do this; two objects in the environment can be attached to each other in various ways. Latest developments in the CRAM-Bullet interface allow loosely attached objects, simulating, for example, the transport of multiple items on a tray, or moving a surface with several objects on top. It is possible to both pick an item from the tray and transport the tray with all objects on top of it. This functionality is used when simulating holders for the airplane parts. Every holder is rigidly attached to the assembly board and each airplane part loosely put on their

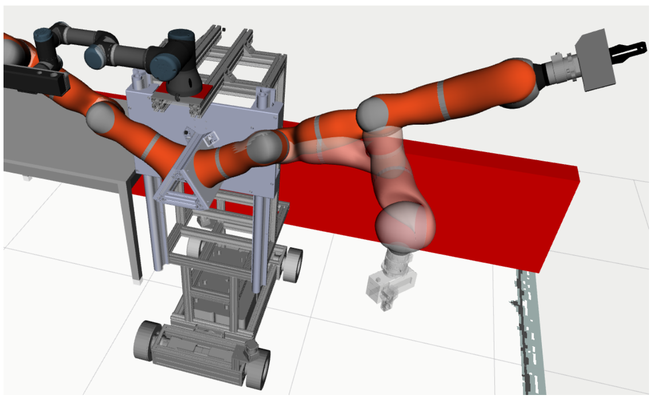


FIGURE 2.5: Visual prediction of Giskard's motion plan in RViz.

respective holder. Moving the assembly board also moves the holders, as well as their parts, in the same direction, as a chain of attachments. Picking a part from its holder attaches the part to the gripper but does not result in moving the holder or any other part.

When the collision shape of two objects are in contact, their point of collision can be retrieved from Bullets simulation easily. When the gripper touches an object in the real world, an estimated contact point is determined in Bullet and compared to the current position of the gripper during contact. The difference between actual and estimated contact point can be used to adjust the touched object.

Adjusting an object's position is one of the main topics in this thesis. Since Bullet is used as belief state for the robot's environment and every movement of the gripper relies on the object's positions within Bullet, these position are essential for a successful assembly.

2.6 RViz

RViz[2] is an interactive visualization tool for ROS software. It is capable of showing complex meshes of objects, environments and the robot itself, but can be easily extended to render custom markers. It also allows several convenient features, e.g. specify the initial estimate of a robot's pose in the environment for the localization algorithm, command navigation goals, measure distances and provides interfaces to implement custom functions. An example of RViz' visualization is given in the top-left image of figure 2.4, where Boxy stands before the assembly, representing its currently simulated configuration. The green boxes on the table are a part of Giskard's collision scene. The airplane parts in the bottom picture of the same figure show each assembly part without Giskard's bounding box. These parts are publishes as CAD-model markers into RViz' scene to visualize Bullet's current belief state.

In this thesis RViz is used for visual monitoring of the current belief state and Giskard's trajectory planner, which is visualized in real time before the robot actually moves, allowing to cancel the process if Giskard's trajectory doesn't seem promising (see figure 2.5). Custom markers published from high-level control in CRAM augment goal poses and trajectories, which encourages rapid prototyping in simulation and in real world.

[2]http://wiki.ros.org/rviz

2.7 Gazebo

For the capability to simulate wrench sensors, I additionally used the Gazebo simulator[3], as Bullet does not have this capability. In general, Gazebo has been a central component in physics simulation for robots in ROS environments, since it offers a pretty accurate joint and collision model through implementation of the Open Dynamics Engine[4] (ODE). In this thesis Gazebo is only used for its capability to simulate wrench sensors, which makes it possible to test the first approach in section 3.2 within an artificial environment.

2.8 CRAM

Coming to the last and most important component, the Cognitive Robot Abstract Machine (Mösenlechner, 2016) or CRAM in short. CRAM is a computational, cognitive framework for developing high-level robot control programs in Lisp. It is used for monitoring robot behavior and rapid prototyping of robot's actions, and combines a high-level descriptive language for action planning of autonomous robots with sophisticated reasoning mechanisms. An integrated Prolog interpreter enriches symbolic reasoning even further, providing the necessary capability to translate descriptive code into sub-symbolic representations without compromising dynamic resolution.

Bullet was included into CRAM by interfacing its C++ library for reasoning about action effects through accurate simulation (Mösenlechner and Beetz, 2013). CRAM's descriptive plans can be easily tested for robustness through collision detection and physics simulation, and new procedures immediately visualized. The interface between CRAM and Bullet provides a broad tool-set to spawn and move objects, but also complete robots, as Bullet can connect several rigid bodies with different types of joints.

In the first place, CRAM provides useful control mechanism to design reactive procedures. Since this framework is to be applied on robots in the real world, failure handling and recovery from faulty states is of grave importance. Also important for robot control is handling processes in different ways. In the following, all mechanisms used in this thesis are explained.

2.8.1 Fluents

Fluents are an elegant way to react upon changes in external input, such as sensor data. Usually, reactive mechanism are achieved through synchronizing multithreaded processes, while certain threads constantly evaluate callbacks from foreign processes. With fluents in CRAM reactive procedures can be designed in convenient ways. A fluent is a data-structure that provides notification services while holding a value which can change over time. They can be instantiated like any other Lisp object. In the code snippet below a fluent is created and stored in a variable.

```
(defparameter *number-fluent* (make-fluent))
(value *number-fluent*)          ;; result: NIL
```

To get and assign a value to this `*number-fluent*` the `value` function is used. Initially, the fluent's value is NIL, because no other value has been given. The following function sets the *number-fluent* to a given number from the input argument.

[3] http://gazebosim.org/
[4] https://www.ode.org/

```
3  (defun countdown-callback (number)
4      (setf (value *number-fluent*) number))
```

As the function's name already implies, some kind of callback is to be set up. Imagine there is a ROS topic called `/countdown_number` that counts down from 10 to 0 every second, then starts again at 10. When subscribing to this topic, each new number can be immediately stored as the fluent's new value.

```
5  (subscribe "/countdown_number"
6             "std_msgs/Int8"
7             #'countdown-callback)
```

This subscriber's sole work here is to call `countdown-callback` every time a new number is published. Since it runs in a background process, the current thread is not blocked. Checking the fluent's value occasionally will now yield the current countdown.

```
8  (value *number-fluent*)         ;; result: 2 (assumed)
9  (sleep 1.0)
10 (value *number-fluent*)         ;; result: 1
11 (sleep 1.0)
12 (value *number-fluent*)         ;; result: 0
13 (sleep 1.0)
14 (value *number-fluent*)         ;; result: 10
```

As long as new message are published on `/countdown_number` the `*number-fluent*` keeps updating its value. While this process goes on, other tasks can be accounted for until the fluent is needed. Because the fluent contains a value, it is called a value-fluent. Instead of waiting a second for each new value, the `pulsed` function can be used, which creates a condition-fluent that fires an event every time a value-fluent changes its value. This function immediately fires an event if the fluent is at least initialized, afterwards it only does when the value changes. In combination with the `whenever` control macro, the above code can be meld into something more elegant. Line 16 is executed every time the `*number-fluent*` is pulsed.

```
15 (whenever ((pulsed *number-fluent*)) ;; blocks thread infinitely
16     (print (value *number-fluent*))) ;; prints ..., 3, 2, 1, 0, 10, 9, ...
```

When a procedure should wait for a certain condition to arise, the `wait-for` macro can be useful. Comparing a value-fluent to another number creates a condition-fluent, which fires an event each time it is true. Such a junction is called a fluent-network. Reacting to condition-fluents is the main goal when working with value-fluents to create reactive procedures.

```
17 (wait-for (= *number-fluent* 0)) ;; blocks thread until fluent is zero
18 (print "Liftoff.")
```

For this demonstration the two previous mechanisms are combined with the `pursue` macro, which allows execution of multiple threads in parallel. When one of the threads terminates, the others are interrupted and terminate as well.

```
19 (wait-for (< *number-fluent* 5))          ;; blocks thread until fluent is below 5
20 (pursue                                   ;; executes two threads simultaneously
21     (whenever ((pulsed *number-fluent*))  ;; blocks infinitely
22         (print (value *number-fluent*)))  ;; prints 4, 3, 2, 1 and maybe 0
23     (and (wait-for (= *number-fluent* 0)) ;; blocks until fluent is zero
24          (print "Liftoff")))              ;; prints, then terminates both threads
```

New value-fluents can also be created from an already existing value-fluent. These new value-fluents are updated with every pulse of the original fluent. To create such a fluent-network, the original value-fluent is used as argument in a function, supported by the `fl-funcall` macro.

```
25 (defparameter *negative-number-fluent* (fl-funcall #'- *number-fluent*))
26 (whenever ((pulsed *number-fluent*))          ;; blocks thread infinitely
27     (print (value *number-fluent*))           ;; prints ..., 2, 1, 0, 10, 9,...
28     (print (value *negative-number-fluent*))) ;; prints ...,-2,-1, 0,-10,-9,...
```

Such fluent-networks can also be created on-the-fly, since a fluent is always administered by CRAM in the background. Binding fluents to parameters is usually done to maintain persistence, e.g. when initially binding a fluent to a topic's value. In the following is a last example with a different data-type. Since most sensor data doesn't consist of a single value and this thesis works mostly with wrench sensors, the following fluent contains the `geometry_msgs/WrenchStamped` message type. To compare such structures, they must be decomposed first. The following function returns the sum of the absolute value of all directional forces from such a message.

```
29 (defun aggregate-force (wrench-message)
30     (with-fields ((f-x (x force wrench))
31                   (f-y (y force wrench))
32                   (f-z (z force wrench))) wrench-message
33       (+ (abs f-x) (abs f-y) (abs f-z))))
```

As previously shown with the `fl-funcall` macro in line 25, a fluent containing wrench data can be called equivalently to create a new value-fluent. In this example the fluent is named `*wrench-fluent*` and the thread is blocked by `wait-for` until the sum of all forces is above 1Nm.

```
34 (wait-for (> (fl-funcall #'aggregate-force *wrench-fluent*) 1))
35 (print "Contact detected.")
```

Yes, this is the mechanism to detect contact. Later, in section 3.2, this procedure is further elaborated.

2.8.2 Designators

Designators are essential constructs for designing abstract plans. Instead of discrete values they describe actions, motions, objects and locations as symbols, containing general information about their purpose.

A designator is interpreted by the Prolog engine, gathering all missing information from Bullet, the robot-description and its current state, environmental constraints and many more. The interpretation of a description continues until every part, and its inferred designators within, are resolved into discrete arguments. As typical for Prolog, multiple solutions for one description are possible, e.g. when finding a suitable position nearby the assembly board, where the gripper can reach a certain object. In such cases, if the first solution doesn't suffice, the next one is probed until it does, which is simulated in Bullet before real-world execution. This procedure is called 'fast projection' which rapidly teleports the robot into several configuration by a simple inverse-kinematic solver, to check Bullet for feasibility. In this thesis, however, Giskard is used for motion planning, therefore no projection is done.

Three major types of designators are distinguished: Location, Object and Action. There are also designators called Motions, which are the termination point of Prolog's interpretation for Action-Designators, leading directly to robot-specific functions. Because each designator is interpreted by, Prolog need to know how to distinguish between descriptions (keywords) and values. Those parts of a designator with discrete information have a question-mark as first character, to signalize Prolog that these values can interpreted immediately.

Motion Designators are named after their most frequent use, but can refer to any kind of low-level platform-dependent function. These lowest-level designators are the result of higher-level designators being completely resolved by the Prolog interpreter, therefore each Motion Designator only contains discrete information. Each such designator is mapped by so-called process modules to platform-dependent implementations. In this thesis, all process modules for controlling robot movement are directed to the CRAM-Giskard interface, while information about objects and locations is provided by Bullet, flavored with predefined parameters in CRAM. Suitable grasping position, for example, are described w.r.t. each object-type. When transitioning from one platform to another, only the process modules' endpoints need to be implemented, which are the interface between CRAM and platform-specific implementations.

Object Designators cover the description of things in the world. Every object can be described by their name or certain features. The following will yield any object of the type bolt, of which at least five exist in the assembly scenario.

```
(an object
    (type :bolt))
```

To get one specific bolt, its name can be used. Here the third bolt is searched for.

```
(an object
    (name :bolt-3))
```

The two designators above are only descriptions, which need a Motion Designator to resolve them into discrete values. Instead of actually perceiving each object by visual classification, their representation is fetched from Bullet's belief state with the following Motion Designator.

```
(a motion
   (type world-state-detecting)
   (object (an object
               (type :bolt))))
```

Furthermore, objects from certain locations can be acquired, e.g. on the assembly board. For this, several designators can be nested.

Location Designators are required for such spacial reasoning. They are capable of describing poses, which are interpreted immediately, and regions, which depend on the task at hand.

```
(an object
    (location (a location
                 (on (an object
                         (name :assembly-board))))))
```

When placing an object onto the assembly board, only the inner location designator is needed to generate a region of possible poses. CRAM then tries to place the object on any possible position within the specified region through fast projection. Since in this thesis every part is meant to be assembled on one specific object, no such descriptions are used. Instead, a location can be determined by a certain pose. In this example a static pose is declared in a local variable a prior.

```
(let ((?goal-pose (make-pose-stamped "map" (ros-time)
                                     (make-3d-vector 1 2 3)
                                     (make-identity-rotation))))
  (a location
     (pose ?goal-pose)))
```

Action Designators are required from here on, be it to move certain robot parts into position, call image processing, opening/closing the gripper or similar processes. The latest design of action designators was published in (Kazhoyan and Beetz, 2017) and further developed in (Kazhoyan and Beetz, 2019). Moving the robot's base in front of the assembly-board is designed like this.

```
(an action
    (type going)
    (target (a location
               (in-front-of (an object
                                (name :assembly-board))))))
```

For this thesis three new actions are developed, which all move the gripper under different conditions: reaching, pushing, retracting. The circumstances define how Giskard calculates its trajectory towards the given goal. The `reaching` action is used to bring the gripper in a general position nearby its actual target. During this movement to one specific pose it depends on the task at hand, whether certain joints need to be restricted from moving. In the example below, the base and torso joints are restricted, such that only the arm is considered by Giskard. Such restrictions are required in the second approach in section 3.3. Also, any potential collision should be avoided between the gripper's current position and the goal (line 8).

```
1 (an action
2     (type reaching)
3     (pose ?start-pose)
4     (constraints ("odom_x_joint"
5                   "odom_y_joint"
6                   "odom_z_joint"
7                   "triangle_base_joint"))
8     (collision-mode :avoid-all))
```

When `pushing`, however, the gripper is expected to come in collision with other objects, therefore all collisions are allowed, but the base is still restricted. It would also be possible to only allow collision between specific objects, but since multiple parts of the airplane can overlap, determining which collision to allow is hard. This action receives a trajectory of poses, which the gripper approaches sequentially.

```
(an action
    (type pushing)
    (left-poses ?trajectory)
    (constraints ("odom_x_joint"
                  "odom_y_joint"
                  "odom_z_joint"))
    (collision-mode :allow-all))
```

Lastly, `retracting` from a task is done with full freedom of base movement. Even though the gripper's bounding box is probably still in collision with some other shape in Giskard's collision scene, full avoidance wanted. This causes Giskard to first remove the gripper out of the collision safely, before approaching the given retracted pose.

```
(an action
    (type retracting)
    (pose ?retracted-pose)
    (constraints nil)
    (collision-mode :avoid-all))
```

Per action, the constraints and collision preference is implied through the Prolog interpreter. Therefore they won't be mentioned in the implementations within section 3. In the next section fluents and designators are brought together in CRAM's failure handling mechanism.

2.8.3 Self-Recovering Plans

Some CRAM macros like `pursue` enable multi-threading, others cause interruptions to the main thread. The common-lisp error handler can't proficiently account for such conditions, therefore CRAM offers its own failure-handling strategies. An example of CRAM's failure handling mechanism is shown in listing 1.

```
1  (with-retry-counters ((retries 100))
2      (with-failure-handling
3          ((force-detected (e)
4              (print e)
5              (do-retry retries
6                  (setf ?trajectory (calculate-new-trajectory-from ?trajectory))
7                  (an action
8                      (type reaching)
9                      (pose ?start-pose)
10                 (retry)))))
11         (pursue
12             (perform
13               (an action
14                   (type pushing)
15                   (left-poses ?trajectory)))
16             (and (wait-for (fl-funcall #'sum-forces *wrench-fluent*))
17                  (fail 'force-detected)))))
```

LISTING 1: Self-recovering Plan example code: Main section to achieve (11-17), including gripper movement (12-15) with potentially raising a condition (16-17). Condition-handler (3-10), including changing the trajectory (6), retracting the gripper (7-9) and retrying the main section (10).

with-failure-handling works mostly like any try-catch mechanism in languages like Java or C++, except it can handle multi-threaded processes and complex macro dynamics when operating with fluents and actions. Its implementation is based on the common-lisp case-handler. What makes it shine is the ability to invoke restarts on failed sequences. This enables a procedure to recover from the failed state before trying again. In this thesis the retrying behavior is exploited by reacting to contact events and changing the sequence's parameters before invoking it again. Listing 1 shows a rudimentary example for the `with-failure-handling` mechanism.

In contrast to failure-handling in other languages, the catch-block is defined before the critical sequence, so invoking the procedure in said listing jumps to line 11 first. It tries to move the gripper along a trajectory of poses. If a contact event is detected the `force-detected` condition is handled, then the main sequence is invoked again. The whole procedure only terminates when the gripper has passed all poses of the trajectory without causing collision.

with-retry-counters offers to terminate procedures after a number of retries. If a procedure takes too many attempts without success it can be terminated. After a maximum of 100 attempts (line 1) the procedure in listing 1 doesn't handle the `force-detected` condition anymore. Usually such procedures are nested within more general plans, which can react upon the propagated error themselves on a higher level, e.g. by moving the base to a more promising position before invoking listing 1 again.

Chapter 3

Methods and Implementation

Assembly tasks are heavily dependent on precise knowledge of the robot's surroundings, especially in an uncertain environment. Robots are limited in precision, because they can only be as accurate as digital encoding simulates the actual robot. Performing assembly tasks has been present for decades, concentrating on the transition from human demonstration to programmable procedures (Takahashi and Ogata, 1992) and is still a contemporary topic, now including humans in collaborative scenarios (Hietanen et al., 2020; Sun et al., 2020). It is the procedure of putting two objects together, usually done by manipulating one object, held in a gripper, and moving it on or in an other object. This is easy for a human, because we feel each change in movement of the objects we hold in our hands and see its orientation, can blindly find keyholes with our fingertips and feel when the key fits. We can even predict if two object could fit together just by looking at them. For robots however, it is an immensely difficult task.

Usually a robotic arm like the KUKA LWR series or Universal Robots is attached rigidly to a table when executing precise object manipulation (Khansari-Zadeh, Klingbeil, and Khatib, 2016; Inoue et al., 2017; Stelter, Bartels, and Beetz, 2018). In this thesis the humanoid robot Boxy is used in a real-world environment (see figure 2.1). The robot can move holonomically (Kyung-Lyong Han et al., 2009), while estimating its position with one LIDAR sensor in the base by using Augmented Monte Carlo Localization (Hiemstra and Nederveen, 2007). It has a 7 degree of freedom KUKA LWR-4 arm, attached to a prismatic torso joint. Especially the moving base makes maintaining precision difficult. Since the robot's position can only be estimated, be it by LIDAR, acoustic, 2D or 3D image processing, the data and its interpretation provides accuracy to a certain degree, depending on their respective quality.

In this thesis Boxy is tasked to assemble a toy airplane from Battat[1] which is included in the YCB dataset (Calli et al., 2015). The airplane parts can be seen in figure 3.1[2]. Using the YCB dataset and assembling this type of toy airplane is viewed as a benchmark test for contemporary assembly tasks, because is provides objects of different shapes, color, weight and rigidity, making the task more or less difficult, depending on the subset of parts considered in experiments, and the gripper used for manipulating them. Each plane part is held by a custom 3D-printed holder. Compared to peg-in-hole tasks this scenario is directed at diverse approaches on every-day shaped objects. The airplane's parts are placed on an MDF board, where the 3D-printed holders are screwed upon, holding the parts at positions where Boxy, our robot, can reach them.

[1] https://battattoys.com/product/battat-take-apart-airplane/

[2] Downloaded on 03.11. from https://battattoys.com/wp-content/uploads/2018/12/BT2517_ToyPlanePieces_02.jpg

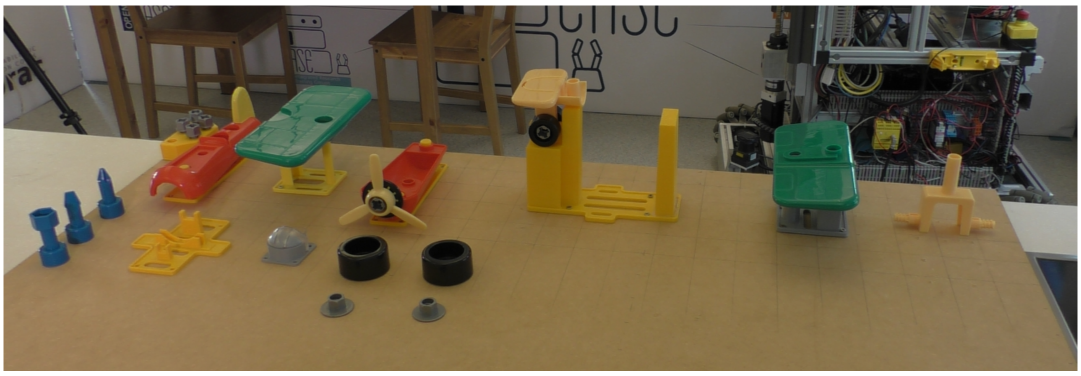


FIGURE 3.1: From left to right: Screwdrivers, vertical plane holder, bolts, upper-body, top-wing, windshield, nuts, front wheels, under-body with grill and propeller, rear-wing on horizontal plane holder, bottom-wing, chassis.

To achieve this task the robot must know of its surroundings and positions of the airplane's parts through a belief state, which is a simulated representation of the real world. Unfortunately, even with an immensely precise believe state with respect to the real world, intricate tasks are still hard to accomplish if the robot's estimated state lacks precision.

The first idea to overcome inaccuracy in uncertain environments for intricate tasks is similar to a reverse peg-in-hole task, since the airplane's assembly mostly consists of hollow object being put on holders and pegs. If an object is already held in the gripper and then assembled onto its target, contact forces occur which can be reasoned upon. Many before had this idea, basically since peg-in-hole tasks came to life several decades ago [3]. Nevertheless, an approach was designed to adjust the gripper depending on what forces occur on collisions during assembly, to incrementally adjust the gripper into correct position. The procedure works on paper and in simulation, but is not reliably applicable in real-world. Only during real-world experiments several issues manifested themselves: the influence of friction between the objects in contact, as well as their slipping in the gripper and the spring-like behavior of the LWR-4+ arm joint-impedance controller (see section 4.3.1).

Therefore a second approach is designed. Instead of adjusting the gripper's position, the idea is to find the airplane parts by touching them. Even if a belief state is pretty accurate it only simulates relations between objects in the environment, but the main problem in this thesis is actually the robot's imprecision in the environment. By touching the airplane parts before manipulating them, their position is adjusted in the belief state with respect to the gripper, hence the robot. If an object is touched from several sides its position is estimated even better. Instead of simulating an environment with respect to a fixed point in the real world, the environment is adjusted such that the robot is the point of reference. Manipulating objects precisely can only be done when their position is precise with respect to the robot.

The belief state of choice in this thesis is a physics engine called Bullet Physics Library (Coumans, 2015) coupled with the Cognitive Robot Abstract Machine (CRAM)

[3](Bruyninckx, Dutre, and De Schutter, 1995; Newman, Zhao, and Pao, 2001; Sharma, Shirwalkar, and Pal, 2013; Inoue et al., 2017; Park et al., 2020; Jiang et al., 2020; Liu et al., 2020)

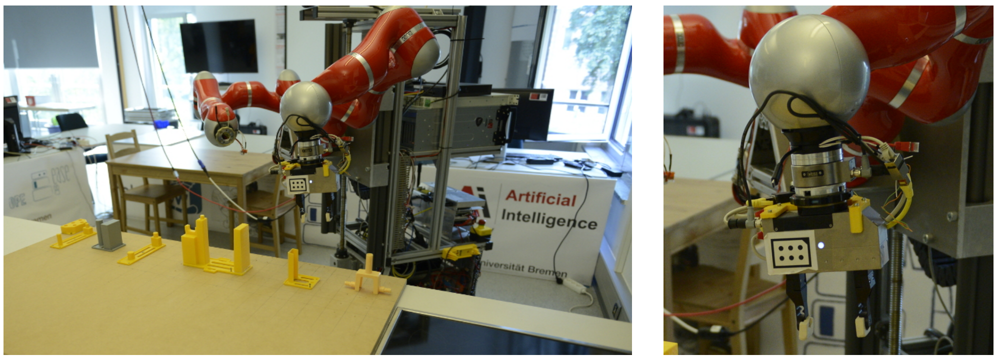


FIGURE 3.2: Left: Boxy in front of the assembly board. Right: Gripper and force-torque sensor.

in which all procedures presented in this thesis are developed in. The robot manipulates all airplane parts with a gripper attached to the LWR-4+ arm's end effector. Moving the gripper to cartesian poses is planned by Giskard, who calculates joint velocities over time-sequences for each joint involved in the gripper's movement, to move joints from multiple controllers in parallel. With Giskard a cartesian goal for the gripper is rarely out of reach, because Giskard involves the base, torso and arms when finding joint trajectories for all controllers involved, to move each component at once. To move the gripper without collision Giskard has a collision scene, which is an exact copy of the objects in Bullet and continuously synchronized with Bullet's belief state. This collision scene contains bounding boxes for each object to calculate the gripper's movement around them.

Bullet receives the robot state including localization through ROS, enabling fast prototyping when a robot is in simulation, while in real-world it updates its internal robot state with an estimate of the robot state from each joints controller. RViz is used to visualize Bullet's belief state, Giskard's collision scene, and the current robot state first-hand from the joint controllers.

Since Bullet is the standard physics engine for CRAM its functionality has grown with the development on CRAM and now provides all necessary capability as belief state regarding spacial reasoning, collision detection and attaching two objects together. Grasping an object attaches it to the gripper, moving both simultaneously, which inherently extends collision-avoidance in Giskard by including the held object into its collision scene. The holders are attached to the board, such that the simulated holders move, when the board moves, therefore the objects attached to the holders as well. It is possible to define attachments loosely, such that objects can be moved from holders without moving the holder itself. Collision detection in Bullet is done by checking for overlapping bounding boxes. Determining the contact point between two objects is done with bounding boxes as well. To increase accuracy in detection of contact points between two objects, their mesh representations are decomposed into a collection of single collision meshes. What influence this has is explained in section 3.3.1.

Using the existing implementations from the previous chapter, the following part focuses on the core idea of solving this thesis' proposed problem, utilizing force data to improve accuracy in assembly tasks. Two approaches are taken to achieve this: the first one incrementally adjusts the gripper's position while pushing a held object onto the target, the second one adjusts the target's position in the internal world state by touching it from different angles. They are tested and evaluated through a series of assembly tasks, which are explained in the Experimental Evaluation of chapter 4.

Both approaches react to force data while moving the gripper to the target. Before any force-torque data can be reasoned upon, there needs to be some data to begin with, which is explained in the first part of this chapter. Afterwards the first approach is presented, involving a classification heuristic that determines, how to adjust the gripper's position in order to successfully reach its goal position. This is achieved by evaluating the force data perceived while pressing against the target object with another object held in the gripper. In the second approach force data is used to detect a touching event between the gripper and the object, after which the object's pose is adjusted based on the gripper's position while in contact. The position of all objects interacted with is uncertain to a degree, which is further elaborated in chapters 4 and 5.

3.1 Sensor Communication

A Weiss KMS40 force-torque sensor is used. In physics a force and torque vector can be assembled into a screw, a six-dimensional vector constructed from a pair of two tree-dimensional vectors representing linear and angular velocity, commonly known and further called **wrench**. The wrench sensor is attached as a fixed wrist joint between the last link of the left LWR3 arm and the end effector, a Weiss WSG50 gripper (see figure 3.2). All object manipulation is done solely by the WSG50 gripper, while the KMS40 wrench sensor constantly sends data. A ROS node publishes the sensor data onto a ROS topic. The communicated message `geometry_msgs/WrenchStamped` is shown in listing 2.

```
1  std_msgs/Header header
2      uint32 seq
3      time stamp
4      string frame_id
5  geometry_msgs/Wrench wrench
6      geometry_msgs/Vector3 force
7          float64 x
8          float64 y
9          float64 z
10     geometry_msgs/Vector3 torque
11         float64 x
12         float64 y
13         float64 z
```

LISTING 2: Wrench message: Standard ROS header (1-4). Wrench data (5-13) with directional (6-9) and rotational (10-13) velocities.

The wrench data can be zeroed by calling the `ft_cleaner/update_offset` service which is wrapped in the Lisp function `(zero-wrench-sensor)`. Zeroing the data is important, because the real wrench sensor accumulates a drift over time, especially when the gripper's orientation changes, since its inertial calibration has limited precision. Also, if the gripper holds an object, the object's weight influences the sensor, and releasing the object changes wrench again.

3.1.1 Using the Wrench-Sensor in CRAM

With the sensor data published into the ROS network, any node can subscribe to it. Within CRAM, there is a specific data structure for representing continuously changing data called fluents. When subscribing to a ROS topic a subscriber updates the fluent with each new message, the synchronization is taken care of by the data structure. Detailed explanation of fluents in CRAM is given in section 2.8.1 and in the original work on CRAM (Mösenlechner, 2016).

Some functions in this thesis rely heavily on the wrench sensor. Listing 3 describes a mechanism to verify that the sensor is transmitting data.

```
1  (defun fl-active (&optional (fluent *wrench-state-fluent*)
2                              (sample-rate *fl-default-sample-rate*))
3    (declare (type cpl:value-fluent fluent))
4    (let ((init-pulse-passed nil)) ;; for better readability
5      (pursue
6        (whenever ((pulsed fluent))
7          (if init-pulse-passed
8              (return T)
9              (setf init-pulse-passed T)))
10       (progn (sleep sample-rate)
11              nil))))
12
13 (defun fl-gate (&optional (fluent *wrench-state-fluent*)
14                           (sample-rate *fl-default-sample-rate*))
15   (declare (type cpl:value-fluent fluent))
16   (unless (fl-active fluent sample-rate)
17           (error "The fluent ~a is inactive." fluent)))
```

LISTING 3: Wrench-fluent pulse monitor to determine the wrench-sensor's activity. Function fl-active (1-11) returns T if the fluent receives data. This is done by checking the fluent's pulse. fl-gate (13-17) raises an error if the fluent is dead.

If the sensor dies, the robot might cause some serious damage, because a procedure might wait for a collision event to occur, but would never get the contact event to react upon. Therefore a safety precaution is implemented, the `fl-gate`. Every fluent emits a pulse when its value is updated, like a heartbeat. The implementation in `fl-active` (lines 1-11) listens to a fluent's pulse to determine if it is still alive, `fl-gate` uses this mechanism to throw an error if the fluent is dead (lines 13-17). Since these mechanisms are mostly used to check if the wrench sensor is sending new data, their default input argument is the `*wrench-state-fluent*`, although it can be used for any kind of value-fluent.

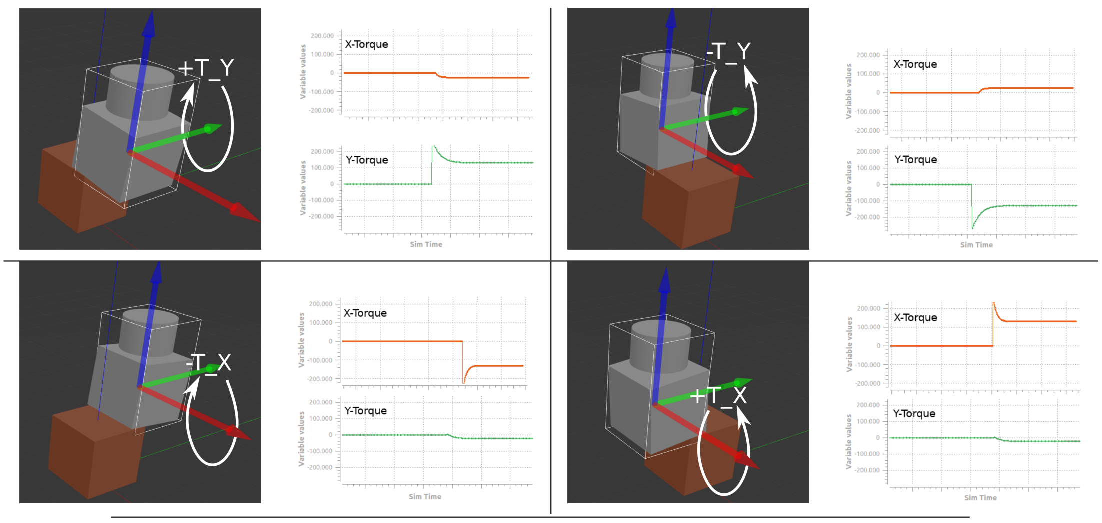


FIGURE 3.3: Strongest torques based on gripper offset.
Top-left: +X offset: +Y torque. Top-right: -X offset: -Y torque.
Bottom-left: +Y offset: -X torque. Bottom-right: -Y offset: +X torque.

3.2 Approach 1: End Effector Pose Adjustment

This is the first approach taken to correct inaccuracy when assembling two parts together. One object is held in the robot's gripper, while the robot is tasked to put the object onto another one.

Through this approach five different pairs of objects are assembled, all by putting a held object downwards onto a target. First the held object is positioned above the target, then it moves down to the assembly's goal position. During downward movement contact between the held and target object is expected if the gripper's position is not accurate along the X-Y plane. As explained in (Bouchard et al., 2015) each two colliding objects yield torques with relation to their point of contact and applied linear force. In Bouchard's work the held object changes orientation due to frictional forces while applying linear velocity upon them, thus torques are applied to resist the objects urge to rotate. Compared to this approach, pushing two misaligned objects together results in rotational movement of the object held, which is transferred to the gripper and thus the wrench sensor. Depending on the direction of misalignment the two object's contact point differs, therefore different torque emerges.

During contact the torques are evaluated, then the gripper retracts from contact by moving upwards. It repeats moving into contact and back again, until enough contact forces are gathered, before adjusting the gripper's pose against the most frequenting classified offset. Then this process starts again, classifying new contact forces with the adjusted gripper. After several repetitions the gripper incrementally adjusts to a position, which eventually allows successful assembly.

3.2.1 Torques in Simulation

Before heading into the real world, this scenario is simulated in Gazebo (described in section 2.4) as visualized in figure 3.3.

The Gazebo simulation models the robot's arm wrist (cylinder) as kinematic body, unaffected by forces, an object held in the gripper (upper cube) which **is** affected by force as dynamic body, and a static collision object (lower cube) is unaffected again. The wrist and gripper including the object are connected via a fixed joint containing a simulated wrench sensor. When the wrist is moved, the upper cube (gripper & object) moves as well, but when the upper cube touches the static bottom cube, only the upper cube moves and presses against the wrench sensor.

In each of the four images the object in the gripper is pressed against the lower cube with offsets in four directions. When comparing the corresponding torques around X and Y with the offset, each case yields easily distinguishable results. In each graph, only torque around X and Y is shown, because linear forces and torque around Z are not as strongly affected as X and Y torques.

The bottom two images in figure 3.3 depict offsets of the gripper along positive and negative Y with the corresponding changes in torque around X and Y. In both scenarios the X torque (T_X) is stronger than around Y (T_Y), with a negative T_X in the left, and positive T_X in the right image. Opposed to that are the top two scenarios, where T_Y is stronger when the gripper has an offset along the X axis. This gives the idea, that a heuristic can be modeled to classify the general direction of an offset, based on torques.

3.2.2 Heuristic Classifier

With the observation made in the previous section 3.2.1 a simple heuristic is developed. Its goal is to determine in which direction the end effector needs to be adjusted on the X-Y plane when approaching the goal position downwards, in order to successfully put one object onto another. The heuristic is shown in listing 4 and takes the `*wrench-state-fluent*` value as input argument, which contains the current data from the wrench-sensor.

```
 1 (defun heuristic (wrench-msg)
 2   (with-fields ((fz (z force wrench))
 3                 (tx (x torque wrench))
 4                 (ty (y torque wrench))) wrench-msg
 5   ;; Gripper must push with reasonable force
 6   (when (> fz 1.0)
 7     (if (> (abs ty) (abs tx))
 8       ;; If Y-torque is dominant, offset in X
 9       (if (< 0 ty)
10           :+x-off
11           :-x-off)
12       ;; If X-torque is dominant, offset in Y
13       (if (< 0 tx)
14           :-y-off
15           :+y-off)))))
```

LISTING 4: Code of the heuristic classifier. Extracts linear force for contact detection, and torques around X and Y (2-4) from the wrench message. Given enough contact force (6) high torque around Y (7) means offset in X (9-11), while torque around X suggests Y offset (13-15). Arrangement of contact force axis and expected torques for the sensor's orientation in the Gazebo demo of section 3.2.1.

Each value returned (e.g. `:+x-off`) is the identifier for the detected offset, based on the angular velocities (torques) around X and Y. Depending on the end effector's orientation during assembly, the order of compared torques must be adjusted. This heuristic only determines an offset in one of four directions, +/-X and +/-Y. If the gripper is misaligned diagonally, only one of the axes is declared for adjustment. Since the gripper adjusts its pose incrementally over several trials, a diagonal misalignment will result in alternating offsets between two axes. The procedure in section 3.2.3 accounts for this behavior.

3.2.3 Procedure

This is the core implementation of approaching the goal pose while constantly adjusting the gripper's trajectory towards the goal. A diagram in figure 3.4 shows the process while the pseudo-code is given in listing 5.

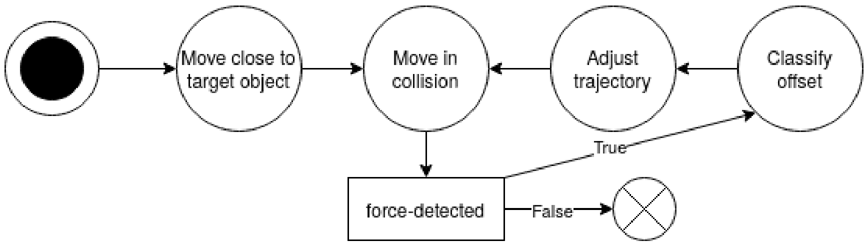


FIGURE 3.4: Process diagram for end effector pose adjustment.

The procedure changes the trajectory towards the goal, based on the heuristic's results during contact events. It takes a trajectory of poses as input and approaches each one sequentially, while cautiously checking for contact events. Any time a collision is detected, the whole trajectory is pushed in the direction recommended by the heuristic classifier. Finally the function ends when the last pose is successfully reached.

The gripper must hold an object in its hand already to assemble it onto the target. This is done by moving the gripper to a pose above to the target object, before moving down towards the target by following a trajectory of poses sequentially, without causing contact forces. Three input parameters are used for this recursive pseudo-implementation: (1) a list of poses (trajectory) to be followed, (2) movement constraints, which are generally used to keep the robot's base still during all gripper movements, (3) and a list of classified results. The results are empty in the beginning and to be filled during each recursive call, by classifying offsets with the heuristic explained in 3.2.2.

The procedure begins by bringing the gripper to the trajectory's first pose. Then it verifies the wrench sensor's correctness (lines 7-10) as described in section 3.1.1. After zeroing wrench data, the gripper tries to follow all poses of the trajectory (lines 11-12) while simultaneously checking for contact (line 13). If a contact occurs, a condition is raised and caught below (line 18). The offset is classified using the current wrench data (line 19). If this classified offset is dominant (over 50% occurrence) among other recorded offsets (line 23), the trajectory's poses are adjusted towards correcting the offset (line 25). After too many results the record is cleaned to start anew (lines 24-25), because finding a dominant result among a lot of data by counting occurrence ratio gets incrementally harder. Then the whole trajectory is recalculated (line 26), whether or not a dominant offset was found. This recalculated

```
 1 retries = 100
 2 def follow_trajectory(trajectory, constraints, results):
 3     first_pose = trajectory[0]
 4     move_gripper(first_pose, constraints)
 5     if retries-- > 0:
 6         try:
 7             if not fl_active(wrench_fluent):
 8                 raise WrenchFluentDead
 9             zero_wrench(wrench_fluent)
10             in_parallel:
11                 for pose in trajectory:
12                     move_gripper(pose, constraints)
13                 if force_on_axis(wrench_fluent, 'f_z') > 1.0:
14                     raise ForceDetected('Contact detected.')
15             open_gripper()
16             move_gripper(first_pose, [])
17             return True
18         except ForceDetected:
19             offset = classify_offset(wrench_fluent)
20             push(offset, results)
21             if len(results) > 3 and is_dominant(offset, results):
22                 results = []
23                 trajectory = adjust_trajectory_against_offset(trajectory, offset)
24             if len(results) > 10:
25                 results = []
26             trajectory = recalculate_trajectory(trajectory)
27             follow_trajectory(trajectory, constraints, results)
28         except WrenchFluentDead:
29             print('Wrench sensor is dead.')
30             return False
31     else:
32         return False
```

LISTING 5: Pseudo code for end effector adjustment.

trajectory starts just above the gripper's current position and goes until the trajectory's last pose, to shrink the whole trajectory closer to contact. Recalculation also includes randomly changing the distance between the current trajectory's poses, in order to apply different strengths when pushing against the target object on each recursive iteration. This brings variation into the caused wrenches, otherwise there would be a high potential for accumulating miss-classified results. With the new trajectory, calculated through offset-adjustment and rearranging the density of poses, this function calls itself (line 27). The recursive procedure terminates, when the last pose is reached within 100 attempts.

3.2.4 Implementation

To implement the procedure in CRAM some additional mechanisms must be established first: Contact detection and reacting to collisions.

Contact Detection

Fluent variables can design conditions, as described in section 2.8.1. The function in listing 6 decomposes wrench data and returns the value of one specified axis. When an object is held vertically from the top, pushing down results in negative force along the sensor's Z axis. Therefore the linear force along Z is used as indicator for

```lisp
(defun force-on-axis (wrench-msg axis)
  (with-fields ((fx (x force wrench))
                (fy (y force wrench))
                (fz (z force wrench))
                (tx (x torque wrench))
                (ty (y torque wrench))
                (tz (z torque wrench))) wrench-msg
    (nth (position axis '(:fx :fy :fz :tx :ty :tz))
         (list fx fy fz tx ty tz))))
```

LISTING 6: Returns the specified axis' value. When called with the `*wrench-state-fluent*` this value changes continuously.

contact in assembly tasks with a gripper pointing downwards. To create a condition for contact detection the following fluent fires an event when the linear force along Z is below -1.0.

```lisp
(fl> -1.0 (fl-funcall #'force-on-axis *wrench-state-fluent* :fz))
```

Zeroing the wrench data before contact detection is imperative to mitigate potential drifts in the wrench data through previous movement of the gripper.

Reacting to collisions

The mechanism in listing 7 resembles the reaction to a touch-event while the gripper follows a trajectory of `?poses` until the last one is reached. As failure handling has been explained in section 2.8.1 it consists of the handling part (lines 1-10) and the sensitive code raising the condition (lines 11-19).

```lisp
1  (cpl:with-retry-counters ((retries 10))
2    (cpl:with-failure-handling
3        ((force-detected (e)
4           (print e)
5           (cpl:do-retry retries
6             (setf classified-offset
7                   (heuristic (cpl:value *wrench-state-fluent*)))
8             (recover-from-collision)
9             (adjust-position-with classified-offset)
10            (cpl:retry))))
11     (pursue
12       (and (wait-for (fl> -1.0 (fl-funcall #'force-on-axis *wrench-state-fluent*
       ↪  :fz)))
13            (fail 'force-detected))
14       (perform
15        (an action
16            (type pushing)
17            (left-poses ?poses))))))
```

LISTING 7: Contact event detection (12-15) and handling (3-10) for downwards gripper orientation and movement (17-20).

The pursue macro embeds (1) listening to the wrench-fluent (lines 12-15) and (2) moving the gripper (lines 16-19), and executes both processes in parallel. Whichever thread terminates first preempts the other thread, meaning, if (2) all goal `?poses` are reached before (1) any force is detected, the `force-detected` condition is never thrown (line 15). Also, if (1) force is detected before the movement is done, the (2) action is interrupted and the condition is handled.

When the `force-detected` condition is raised, it triggers CRAM's failure handling mechanism to react and recover from the situation (lines 5-10), e.g. by evaluating the contact wrench, retracting from the collision, repositioning the end effector and retrying this segment again.

Assembly Plan in CRAM

The CRAM plan on the next page in listing 8 implements the pseudo-code from section 3.2.3. This code combines the mechanism of contact event detection from section 3.2.4 with reactive failure handling from section 3.2.4. Instead of recursive behavior, like in the pseudo-code example, CRAM failure-handling can be utilized (lines 8-23) to retry the main section from lines 39 to 51.

First the main section is explained. The goal of this function is to move the gripper along a trajectory without collision, open the gripper and retract to the start (lines 35-51). To verify the wrench sensor's data, `fl-gate` and `zero-wrench-sensor` are used in lines 33 and 34. Moving the gripper and waiting for contact is done in parallel through the `pursue` macro in line 35. Waiting for a contact event is done with the `*wrench-state-fluent*` in lines 36 to 38, where an event is fired when force on Z is below -1. Moving along the trajectory happens in a `pushing` action-designator, allowing all collision in Giskard's collision scene (line 44).

When a contact event is fired, CRAM's failure handling mechanism is invoked (line 8). During contact the classifier obtains the offset and saves the result in a hash-map called `*results*`, by incrementing the counter for the corresponding offset (lines 12-14). If the offset is dominant, each pose of the current trajectory is transformed in the opposite direction of the offset (lines 18-21), overwriting the current trajectory with these corrected poses. Then the main section is retried (line 32) with the adjusted trajectory.

Eventually the gripper reaches its goal without causing collision, to finally release the object and retract to the beginning (lines 45-51). These movements will not raise an error, regardless of detected forces. A maximum of 100 attempts are allowed for adjusting the trajectory (line 6) after which this procedure fails.

```
(defun follow-trajectory (?trajectory ?constraints)
  (declare (type list ?trajectory)
           (type list ?constraints))
  "`?trajectory' poses along the path to the goal
   `?constraints' for restricting certain joints during movement"
  (cpl:with-retry-counters ((attempts 100))
    (cpl:with-failure-handling
        ((force-detected (e)
           (roslisp:ros-warn (assembly follow-trajectory) "Condition: ~a" e)
           (sleep 0.5)
           (cpl:do-retry attempts
             (let ((heuristic-result
                     (value (fl-funcall #'heuristic *wrench-state-fluent*))))
               (incf (gethash heuristic-result *results*))
               (when (and (>= (total-results 3))
                          (is-dominant result heuristic-result))
                 (reset-results)
                 (setf ?trajectory
                       (mapcar (rcurry #'adjust-pose-with-heuristic
                                       heuristic-result)
                               ?trajectory)))
               (when (> (total-results) 10)
                 (reset-results))
               (setf ?trajectory (rearrange-trajectory ?trajectory)
               (let ((?starting-pose (first ?trajectory)))
                 (perform
                  (an action
                      (type retracting)
                      (pose ?starting-pose)
                      (constraints ?constraints)
                      (collision-mode :allow-all))))
               (cpl:retry)))))
      (fl-gate *wrench-state-fluent*)
      (zero-wrench-sensor)
      (pursue
        (and (wait-for
               (> -1.0 (fl-funcall #'force-on-axis *wrench-state-fluent* :fz)))
             (cpl:fail 'force-detected :description "Contact detected."))
        (perform
          (an action
              (type pushing)
              (left-poses ?trajectory)
              (constraints ?constraints)
              (collision-mode :allow-all))))
      (open-gripper)
      (let ((?start-pose (first ?trajectory)))
        (perform
          (an action
              (type retracting)
              (pose ?start-pose)
              (collision-mode :avoid-all))))))))
```

LISTING 8: CRAM plan for end effector pose adjustment including fluent controlled failure handling. A trajectory of poses is followed by the gripper to deliver an object, open the gripper and retract. If collision is detected, the trajectory is adjusted.

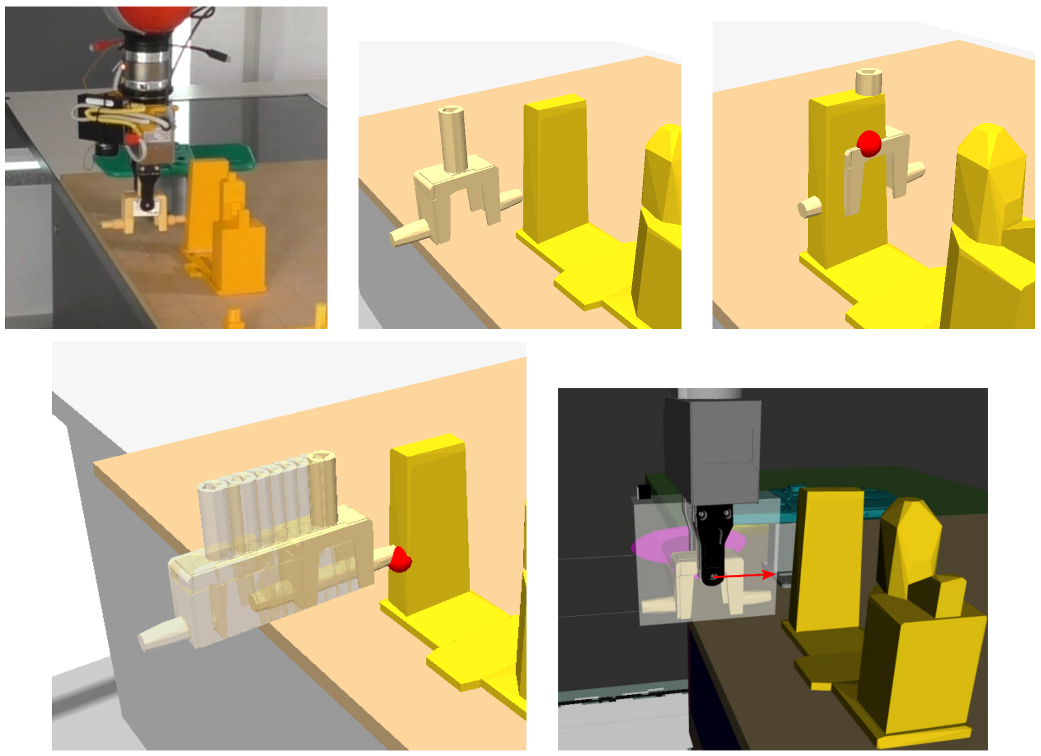


FIGURE 3.5: Adjust holder's position by touching it with a chassis in the gripper. (Top left) Contact in real-world, (top-middle) under-estimating yields no contact point in Bullet, (top-right) over-estimating overlaps the objects, giving wrong contact points, (bottom-left) visualization of pseudo ray casting to estimate the contact point correctly, (bottom-right) chassis in Rviz with directional axis towards the actual contact point on the bounding box' surface.

3.3 Approach 2: Object Pose Adjustment

As explained in the beginning of this chapter, Bullet Physics is used as belief state for the robot's surroundings. Especially important are the airplane parts' positions, which serve as cartesian references for gripper movement during assembly. When picking up an object, Bullet is asked for the object's position and orientation (together called 'pose') to move the gripper w.r.t. the requested pose. The aim of this approach is to identify the actual pose of an object before manipulating it, by touching it in the real world and adjust its pose in Bullet's believe state according to the gripper's position while in contact.

In (Tsujimura and Yabuta, 1989) an object's shape is determined by touching multiple points on their surface. Contemporary work (Kumar et al., 2020) finds the complete shape of an object by continuously moving a tactile sensor array along an object's edges. In this approach, each object's shape is known and their position hard-coded with respect to their holder, which is rigidly attached to the assembly board. But even with positions given, the robot's precision decides about success or failure of assembly tasks, therefore these positions are only as accurate as the robot is. To decrease the object pose's uncertainty w.r.t. the robot, the gripper is moved to an over-estimated pose beyond the object's surface until in contact with the desired object (see 3.2.4). While the gripper is in contact, the real-world contact point is compared to the estimated one from Bullet, in order to adjust the believed pose of the object w.r.t. the gripper.

Only offsets on the X-Y plane are of importance for this assembly task, which is elaborated in section 3.2, therefore the objects are approached only from the sides, not from the top or below. Since each object's shape is known, this information is used to hard-code a promising axis to approach it on, with relation to the object's orientation. These axes are sought to hit a target at its biggest surface, because smaller surfaces are prone to be missed. An object is approached on either the X or Y axis, with predefined offsets depending on the object-type. These offsets are important when manipulating shapes like the airplane parts, because when touching an object, only one contact point is retrieved, and due to inaccuracy it is unclear where this point actually lies on the target's surface. To adjust the object in both X and Y coordinates, offsets for both axes are preset, such that the gripper can touch an object in two separate procedures from perpendicular sides on a promising surface. To eliminate the necessity of estimating an object's orientation, all holders are oriented in 90 degree steps. This arrangement makes it possible to retrieve promising results after a minimum of two contact events.

The actual point of contact is only dependent on the gripper's position while in collision with the object, given by the estimated robot state. On the other hand, the estimated point of contact is retrieved from Bullet, since the robot as well as each object is represented within its belief state. Calculating the difference between real-world and estimated contact yields an offset, with which the object's believed pose can be adjusted. During Experimental Evaluation 4 the difference between actual and believed poses was averaged to around 2cm, but reached up to 3cm in some cases.

3.3.1 Pseudo Ray casting

Bullet's contact point detection between two objects is usually done by checking if their bounding boxes overlap. This is very helpful to rapidly calculate **if** objects are in collision, but is useless when detecting an actual contact point. Most meshes have a concave collision model, representing a bounding box collision shape in Bullet. Two concave collision shapes are not allowed to overlap their bounding boxes, such that a stack of bowls for example would behave like a pile of boxes, instead of neatly fitting into each other. Asking Bullet for contact points between such shapes always yields a point on the bounding box' surface.

To mitigate this problem each objects' mesh is decomposed into a collection of concave collision shapes, called compound-shape. Creating compound-shapes from usual convex-hull shapes is done by Volumetric-Hierarchical Approximate Convex Decomposition (V-HACD), which is implemented for Blender[4]. Depending on the parameters set for this process, the resulting shapes are more or less rougher that its original (see figure 3.6 for an example), but now collision points lie on the object's actual surface instead of the bounding box.

To raise the complexity a bit, not only bare grippers can adjust object poses like this. Moving a held object into collision with another one is needed to find the correct position for an assembly's target in between picking and placing it. In figure 3.5 the airplane's chassis is pushed against the holder to adjust the holder's position. Since differences between real-world and believed contact point are explicitly expected, retrieving the contact points from Bullet between the chassis and holder is

[4]`https://github.com/kmammou/v-hacd`

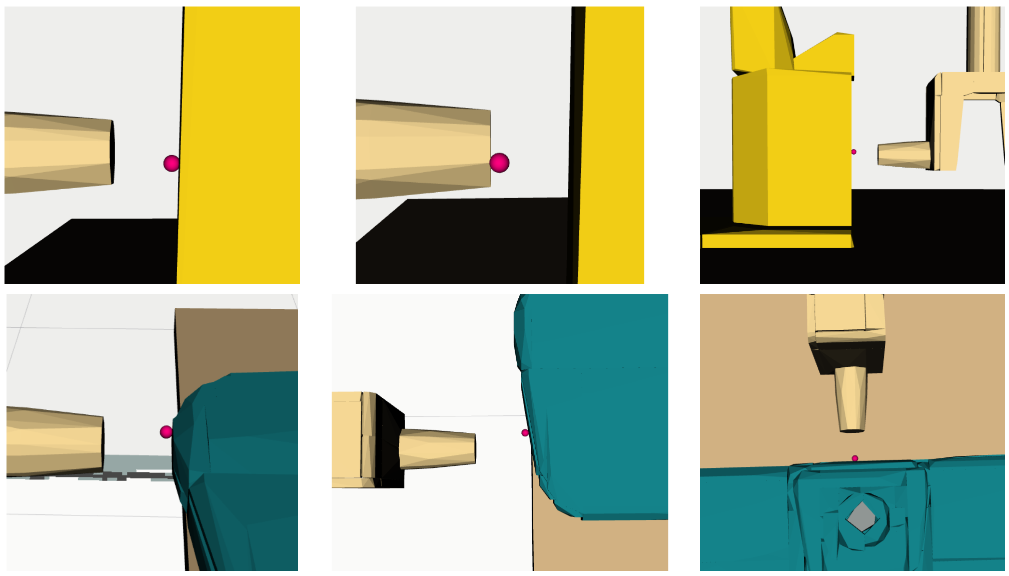


FIGURE 3.6: Contact points (red dots) by pseudo ray casting the chassis against several surfaces. Top f.l.t.r: Contact point on the pedestal, chassis' axis, holder's rear. Bottom f.l.t.r: Contact point on the bottom-wing from the side close to the front, same side closer to the back, from front.

not feasible without further preparation, as can be see in the top three images. Since the believed object's position may not touch or overlap with the target, Bullet's contact point detection yields either none, or unreliable data.

Inspired by the concept of ray casting, a solution for non-colliding and overlapping objects in the belief state was developed, to find a well-estimated contact point. Ray casting in general is used in visualization engines, in which 'To visualize and analyze the composite solids modeled, virtual light rays are cast as probes.'(Roth, 1982). In some way this procedure uses a similar idea to reliably find the estimated collision point between two meshes.

In Bullet an object's mesh attached to the gripper is retracted from the collision along the axis which the gripper took to get into collision, then the mesh is moved towards the target object bit by bit until they are in contact. Without holding an object the gripper's fingertips can be represented in Bullet instead, to use their mesh like an object held. This procedure is visualized in the bottom images of figure 3.6. Instead of casting rays the whole mesh is cast towards the target of interest.

Procedure

In listing 9 is an example of the procedure in pseudo-code. It is called from a procedure explained in section 3.3.2 when the gripper is detected to be in contact with a target object. The input arguments are the held object's name, target name, and the direction on which the gripper is approaching the target. To eliminate changes to the actual belief state, the current Bullet world is copied (line 2). A new vector (direction_step) points towards the direction of approach but is set to a length of 0.1cm (line 3). To pull the held object out of a potential overlapping state, it is retracted by 3cm in the opposite direction (lines 5). Over a range of 6cm (line 6) the held object is projected towards the target in 0.1cm steps (line 11). This movement continues until

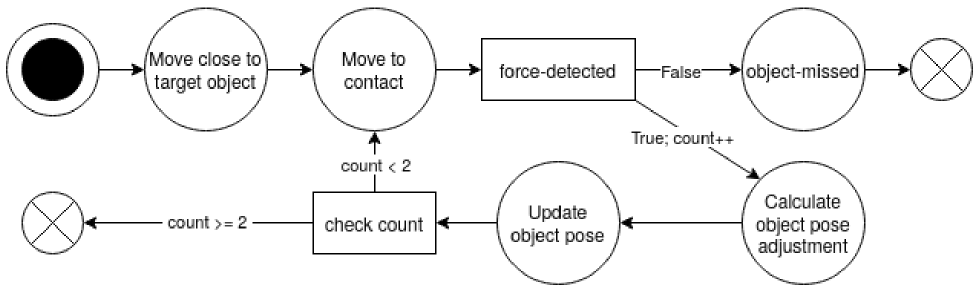


FIGURE 3.7: Process diagram for touching and adjusting the position of an object.

both objects are in contact (line 7-8) which yields a contact manifold, a structure containing both object's bodies as well as their respective collision points. The target's contact point is returned, which terminates the process successfully. Otherwise, an error is thrown if the whole 6cm gone by without any contact, signalizing that the offset must be too high for detecting any contact point.

```
1  def raycast_contact_point(held_obj, target_obj, direction):
2      world_copy = current_bullet_world.copy()
3      direction_step = 0.001 * normalize_vector(direction)
4      max_steps = 60
5      move_object(world_copy, held_obj, direction_step * (max_steps / -2))
6      for _ in range(max_steps):
7          contact_manifold = get_contact_between(world_copy, held_obj, target_obj)
8          if contact_manifold:
9              return get_contact_point(contact_manifold, target_obj)
10         else:
11             move_object(world_copy, held_obj, direction_step)
12     raise OffsetTooBigError
```

LISTING 9: Raycast pseudo-code procedure

3.3.2 Touching the Target

During this procedure (see the process diagram in figure 3.7) the gripper follows a trajectory of poses towards an object on one specific axis normal (along X or Y) until force is detected. While the gripper, with or without holding an object, is pressing against the target, the target's position is updated along the axis on which it is approached. The procedure terminates, when the target has been touched twice on the same axis, to mitigate noisy forces or occasional spikes in the first contact. To adjust and object's pose in X and Y, this procedure has to be called for each axis independently.

Since an object is approached on one axis only, the problem of calculating the difference between actual and estimated position can be broken down to the difference in one value, meaning, depending on the direction of approach only the value of one coordinate is required from the actual and estimated contact point. When an object is approached from the front, the directional vector is [1,0,0], right [0,1,0] and left [0,-1,0]. Directions along Z are not important, because every assembly in this thesis is finished by moving the object downwards onto the target anyway, and touching from behind [-1,0,0] is not possible, due to the arm's limited reach. This is really beneficial for determining the offset.

Actual contact point Each object has a bounding box limited by their furthest expansion from its origin. To determine the actual point of contact, the held object's cartesian position is added to its bounding box expansion towards the direction of approach (see bottom right image in figure 3.5). The resulting point lies on the bounding box' surface of the held object, which is most likely to be in collision. Therefore the offsets of movement per object-type and axis need to be tweaked correctly, otherwise any bigger difference between real and believed position could result in missing the target. From this point on the bounding box' surface, only the direction-dependent value is used. The same procedure goes for using a bare gripper, where the fingertip's dimension is used instead of an object's bounding box.

Estimated contact point With pseudo ray casting in Bullet (section 3.3.1) the held object (or fingertip) is retracted in the opposite direction, then incrementally moved towards the target until in collision (see bottom left image in figure 3.5 and whole figure 3.6). This yields a contact point on the target's surface, which is reduced to the coordinate specified by the axis of approach.

When an object is touched from front, the resulting X coordinates from actual and estimated contact point are subtracted to get the offset along the approaching axis. This difference is applied to the target's position, to transform it towards the actual contact point, resulting in an adjustment in its X-coordinate. Doing the same from left or right will adjust the targets position on the Y axis.

Procedure

The whole procedure is split in two functions, one that moves the gripper into contact, and another that calculates and adjusts the object's pose. Both are explained in pseudo code.

First the touch plan is described in listing 10. It terminates successfully, when the target has been touched twice. As input arguments serve the target object, a trajectory of poses w.r.t. the target object's predefined axis offset, the directional vector, and constraints for the joint controller. The trajectory intentionally over-estimates the goal position beyond the target's surface, to reach it even if the object is further away than expected.

The plan starts off by closing the gripper, whether or not an object is held (line 2). To verify that the wrench fluent is alive, `fl_active` is probed (line 5-6). Then the gripper is moved to the trajectory's first position (line 7). When the movement is done, the wrench sensor data is zeroed (line 8). Now the trajectory is followed while checking for contact forces in parallel, such that the movement terminates as soon as the gripper and target touch (lines 9-14), which then raises a condition. Since the intention of this procedure is **not** to reach the goal pose, it raises an error to signalize that the target was missed after the whole trajectory is achieved (line 15).

When contact is detected, the current gripper position is used to adjust the target along the given axis (line 17), which will be explained in a separate listing below. Since the target is to be touched twice (line 18), on the first run it recalculates the trajectory (line 21). Similar to the mechanism from the first approach of section 3.2.3, the trajectory is changed such that is starts close to the gripper and ends at the same pose, except its poses are slightly denser or wider, which brings variation to the contact forces. Afterwards the new trajectory is followed by recursively calling the touch plan itself (line 23).

```
touched_once = False
close_gripper()
def touch(target_obj, trajectory, direction, constraints):
    try:
        if not fl_active(wrench_fluent):
            raise WrenchFluentDead
        move_gripper(trajectory[0], [])
        zero_wrench(wrench_fluent)
        in_parallel:
            for pose in trajectory:
                move_gripper(pose, constraints)
            if aggregate_force(wrench_fluent) > 4:
                raise ForceDetected('Contact detected.')
                break
        raise ObjectMissed('Pushed to the end but no contact.')
    except ForceDetected:
        update_object_position(target_obj, direction)
        if touched_once:
            return True
        else:
            trajectory = recalculate_trajectory(trajectory, direction)
            touched_once = True
            touch(target_obj, trajectory, direction, constraints)
    except ObjectMissed:
        return False
```

LISTING 10: Procedure to touch an object in pseudo-code

Object pose adjustment during contact

During the touch plan, when the gripper and target are in contact, the target's position is adjusted. The function in listing 11 elaborates calculating this adjustment. As input arguments the target object and direction of approach as 3D-Vector is used, which is assumed to a normal of any axis.

First an axis indicator is set, which contains the directional vector with absolute coordinates (line 2). Depending on whether the gripper holds anything or not, a reference object is set, preferring the held object over the fingertip (lines 3-4). Then the action contact point is calculated, using half the object's bounding box dimensions as 3D vector (line 6) because its expansion starts at the gripper's fingertip position and only reaches half the way towards contact. Also, a little margin is subtracted from each bounding box, because Bullet makes them a little bigger (approx. 2mm) than the meshes actually are. This bounding box vector is rotated into correct orientation (line 7) to then eliminate all its coordinates, except for the axis of interest, leaving a normal vector with the bounding box' expansion in the direction of contact w.r.t. the gripper's fingertip tool-frame (line 8).

The gripper tool-frame position is retrieved by calling TF (line 10), which is a ROS service who maintains poses of all robot's parts. Adding the preciously calculated offset between gripper an contact surface to the gripper's current position, yields a cartesian point on the bounding box's surface w.r.t. the world's origin (line 11). In this point, again, each coordinate is set to zero except for the axis of interest (line 12).

For the estimated contact pseudo ray casting is used, either with the held object or the fingertip (line 14). The resulting point is, again, reduced to a 3D vector with one non-zero entry, on the desired axis (line 15).

The two resulting vectors, actual and estimated, are of the same shape: both with three entries and only one is non-zero. When subtracting the estimated from

```
1  def update_object_position(target_obj, direction):
2      axis_indicator = abs(direction)
3      held_obj = bullet.get_object_held()
4      ref_obj = held_obj if held_obj else robot.fingertip
5
6      ref_obj_bb = (ref_obj.bounding_box_dimensions / 2) - bb_oversize
7      ref_obj_bb_oriented = ref_obj_bb.rotate(ref_obj.orientation)
8      gripper_to_contact = multiply_pairwise(ref_obj_bb_oriented, direction)
9
10     gripper_pose = tf.lookup_transform('map', 'gripper_tool_frame')
11     actual_cp_on_surface = gripper_pose.origin + gripper_to_contact
12     actual_cp_on_axis = multiply_pairwise(actual_cp_on_surface, axis_indicator)
13
14     raycast_cp = raycast_contact_point(ref_obj, target_obj, direction)
15     estimated_cp_on_axis = multiply_pairwise(raycast_cp, axis_indicator)
16
17     move_object(target_obj, actual_cp_on_axis - estimated_cp_on_axis)
```

LISTING 11: Updating an object's pose during contact, in pseudo-code

the actual vector, the result can be applied as transformation to the target's position (line 17). This pulls its surface to the actual collision point in Bullet's belief state.

Contact Detection - Generic

To implement this procedure in CRAM, a different, axis-agnostic contact detection metric is designed. It simply sums all absolutes of linear velocities from the wrench sensor into one value. Compared to force-detection mechanism in section 3.2.4 it doesn't require a specific axis, which makes it easier to handle, since no conversion between 3D directional vector and observed axis is necessary. As the other, axis oriented detection, it is called with the wrench-fluent as input argument.

```
(defun aggregate-force (wrench-msg)
  (with-fields ((fx (x force wrench))
                (fy (y force wrench))
                (fz (z force wrench))) wrench-msg
  (apply #'+ (mapcar #'abs (list fx fy fz)))))
```

Implementation

Following is the implementation of object pose adjustment in CRAM. Only the touch plan is shown in listing 12, since the object pose calculation involves many ROS-specific mechanisms to achieve, and is quite straight-forward anyway.

The main section (lines 23-48) first uses `fl-gate` to verify that the wrench sensor is alive (line 23). After moving the gripper to the trajectory's first position (lines 24-39) the wrench sensor is zeroed (line 30). With the `pursue` macro two threads are executed in parallel: moving the gripper along the given trajectory (lines 35-40), while watching out for contact events (line 31-34). Gripper movement is executed with the `pushing` action designator. Detecting contacts uses the `*wrench-state-fluent*` as conditional-fluent with the `wait-for` mechanic, as explained in section 2.8.1. Both threads raise a condition, so neither can terminate properly.

As explained for the pseudo-code in section 3.3.2 it is intended to terminates the touch-plan from within failure handling. Both conditions are caught independently (lines 14 and 21): while the contact force event `force-detected` invokes the intended sequence, missing an object on the other hand must be handled by a higher-level plan in a different way, e.g. through repositioning the base, or trying a new trajectory. When in contact, the `update-object-pose` procedure from section 3.3.2 is used (line 16) to adjust the target object's position w.r.t. the gripper touching it.

```
 1 (defun touch (?target-obj ?trajectory ?direction ?constraints)
 2   (declare (type keyword ?target-obj)
 3            (type list ?trajectory)
 4            (type 3d-vector ?direction)
 5            (type list ?constraints))
 6   "`?target-obj' name of the object to touch
 7    `?trajectory' list of poses to follow
 8    `?direction' the direction of approach
 9    `?constraints' for restricted joints during movement"
10   (close-gripper)
11   (let ((?start-pose (pop ?trajectory))
12         (touched-once nil))
13     (cpl:with-failure-handling
14         ((force-detected (e)
15            (roslisp:ros-warn (assembly touch) "~a" e)
16            (update-object-position ?target-obj ?direction)
17            (unless touched-once
18              (setf ?trajectory (recalculate-touch-trajectory ?direction))
19              (setf touched-once T)
20              (cpl:retry)))
21          (object-missed (e)
22            (roslisp:ros-error (assembly touch) "Object missed, propagating up.")))
23       (fl-gate *wrench-state-fluent*)
24       (perform
25         (an action
26             (type reaching)
27             (pose ?start-pose)
28             (constraints ?constraints)
29             (collision-move :avoid-all)))
30       (zero-wrench-sensor)
31       (pursue
32         (and (wait-for
33                (fl< 1.0 (fl-funcall #'force-aggregated  *wrench-state-fluent*)))
34              (fail 'force-detected :description "Object touched."))
35         (and (perform
36               (an action
37                   (type pushing)
38                   (left-poses ?trajectory)
39                   (constraints ?constraints)
40                   (collision-mode :allow-all)))
41              (fail 'object-missed
42                    :description "Pushed to the end but no collision."))))
43     (perform
44      (an action
45          (type retracting)
46          (pose ?start-pose)
47          (constraints nil)
48          (collision-mode :avoid-all)))))
```

LISTING 12: CRAM plan for touching an object to adjust its pose. Procedure explained in section 3.3.2.

Chapter 4

Experimental Evaluation

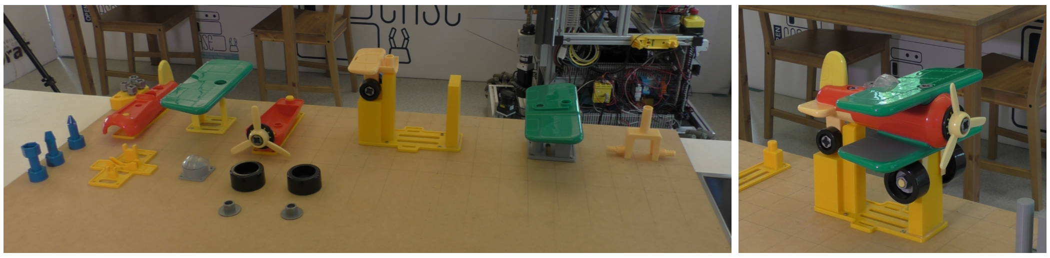


FIGURE 4.1: Left: Setup for the assembly of a Battat toy airplane with parts on holders, screwed on a plate. Back from left to right: Bolts, upper-body, top-wing, underbody, horizontal holder with rear-wing, bottom-wing, chassis. Front from left to right: screwdrivers, vertical holder, windshield, front wheels and nuts.
Right: Assembled toy airplane

This chapter puts the previously explained implementations to the test. First, a general overview of the real-world task is given. Later on, each approach is tested and evaluated in its own way.

Parts of a Battat toy airplane[1] are assembled in the real world to evaluate the two approaches in chapter 3. 3D-printed models of holders are screwed onto an MDF board, which hold the plane's parts in place and makes them accessible for Boxy, the robot. A simulation of this scenario in Bullet already existed, while tests and adaptations for the real world have been developed throughout this thesis. About the difficulties in transitioning from simulation to real world, see section 4.3.

The setup for the plane's assembly is shown in figure 4.1, while the first five steps are depicted in figure 4.2. It starts with putting the chassis onto the horizontal holder, then the bottom-wing is put onto the chassis while the chassis's vertical pin goes through a hole in the bottom-wing. Afterwards the under-body is placed onto the bottom-wing and rear-wing, again guided by vertical pins. The upper-body closes the plane's corpus and a bolt through both body parts and the rear-wing screw them in place. Then the top-wing is placed onto the upper-body and is also screwed tight through both body parts, bottom-wing and the chassis. Now the windshield is attached and screwed to the upper-body's middle region. Lastly the whole plane needs to be placed onto the vertical holder to assemble the front wheels to the chassis and finish the airplane.

[1] https://battattoys.com/product/battat-take-apart-airplane/

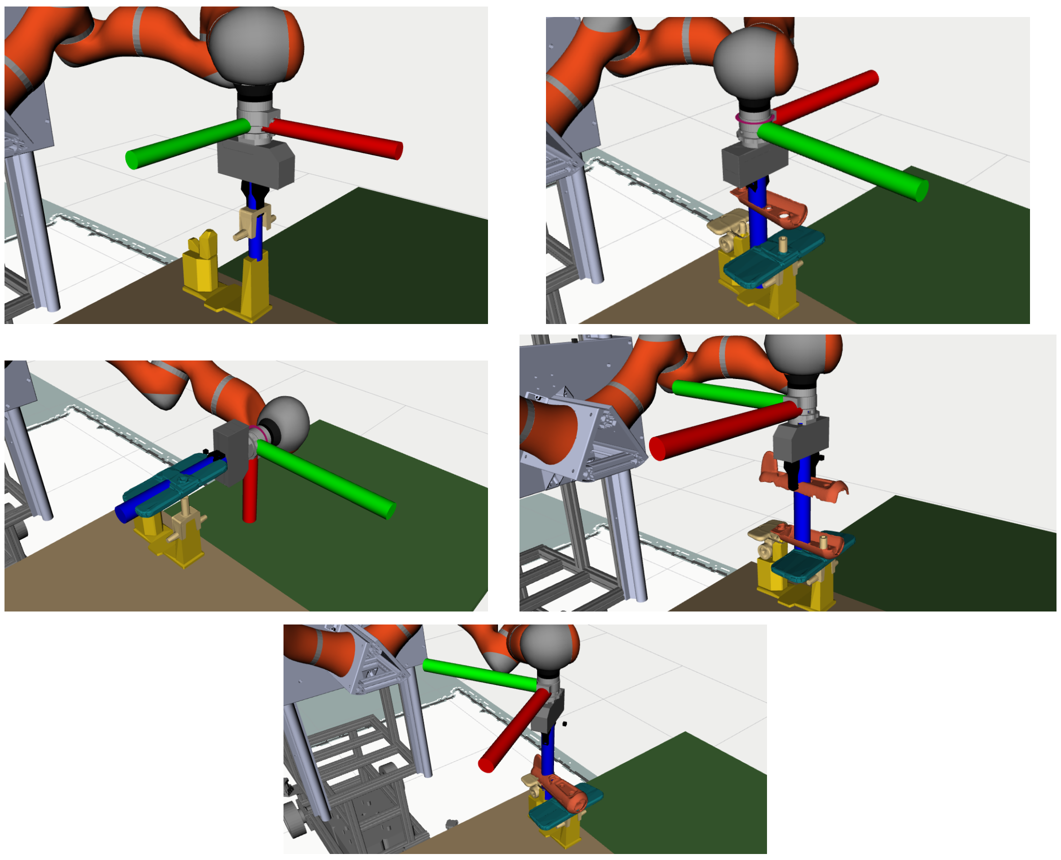


FIGURE 4.2: Wrench sensor orientation during assembly. (Top-left) (1) chassis on holder, (middle-left) (2) bottom-wing on chassis, (top-right) (3) underbody on bottom-wing, (middle-right) (4) upperbody on underbody, (bottom) (5) bolt through rear bodies.

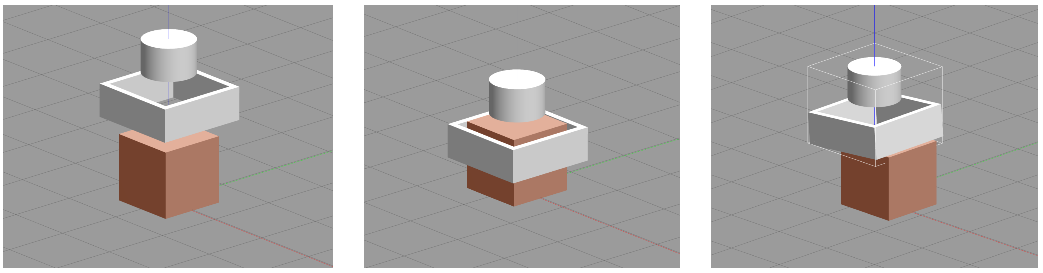
4.1 End Effector Pose Adjustment

The procedure from section 3.2 is tested withing simulation and real-world. Gazebo is used for the virtual environment, whereby only the CRAM plan is needed for execution. In the real world the whole software stack (see section 2) is needed. All tasks for the real-world robot are similar to general Peg-in-Hole tasks, except the actuated and target shapes are switched. Almost every step of assembling the plane puts a hollow object onto a socket, like putting the wheel of a car on an axis.

First the procedure of end effector adjustment is evaluated in Gazebo (see section 3.2.1). Then the heuristic's quality is determined by testing it with real-world contact forces per assembly scenario.

4.1.1 Simulation in Gazebo

To verify the procedure of end effector adjustment in section 3.2 the approach is tested in the Gazebo simulator. In this simulation a hollow object is held in the gripper and put onto a target box, as shown in the top images in figure 4.3. The goal of this evaluation is to see, if the procedure itself is successful before application in the real world, therefore the H_0 hypothesis is, that the procedure generally fails to put the actuated object onto the target. This scenario is similar to most of the assembly steps, but especially the first one: putting the chassis onto the horizontal



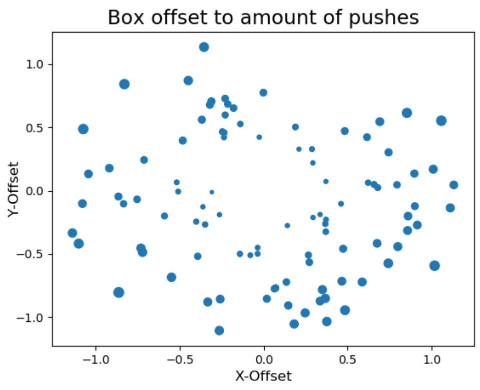


FIGURE 4.3: (Top) Gripper holding a hollow object and target box used in a reversed peg-in-hole task in the Gazebo simulator. (Top-left) actuated object held above the target, (top-middle) successful assembly, (top-right) actuated object in collision, resulting in a wrench to reason upon. (Bottom) Target box's offsets in X and Y over 100 tests. The point's size relate to the amount of pushing the actuated downwards, before successfuly placing it onto the target.

holder. A cylindrical shape resembles the arm's wrist, while the hollow box below simulates the gripper holding an object.

To retrieve contact forces a wrench sensor is simulated between gripper and wrist. Since it is a joint, there is no collision shape to show. This setup is already described in chapter 3.2.1, where figure 3.3 depicts wrenches during contact. The target object is standing on the ground. A Gazebo plugin fetches the simulated wrench data, filters it with a moving average over the past second, and publishes the filtered data on a ROS topic. Compared to the real-world wrench sensor, this data here doesn't need to be zeroed, since it is always stable and doesn't accumulate drift. Also the `fl-gate` isn't necessary, since the simulated sensor never dies. The gazebo_ros interface allows to move gripper and target through service calls. Except for these conveniences, the procedure from section 3.2 is applied as is.

For each test the target is placed at a random location on the X-Y plane. To create random positions, a vector of arbitrary magnitude between 0.3 and 1.2 meters is aligned with the X-Y plane, then rotated at an angle between 0 to 2PI around the world's Z axis. The resulting vector gives the target's offset from the origin. Agnostic of this position, the actuated object is held at X=0, Y=0 and raised above ground (top left image in figure 4.3).

After the above setup the gripper moves down, which brings the actuated object into contact with the target, if not aligned correctly (top-right image in figure 4.3). After one second, when the wrench data reaches a stable curve, contact forces are

TABLE 4.1: Results for end effector adjustment tests in Gazebo. Avg.: average, med.: median, #Push: amount of down-pushes, Class.: classification.

	Time total	**Time avg**	**#Push min**	**#Push max**	**#Push avg**
Results	5236 sec	53 sec	8	68	33.714

	#Push med	**Class. after #Push avg**	**Class. after #Push max med**
	33	4.278	5

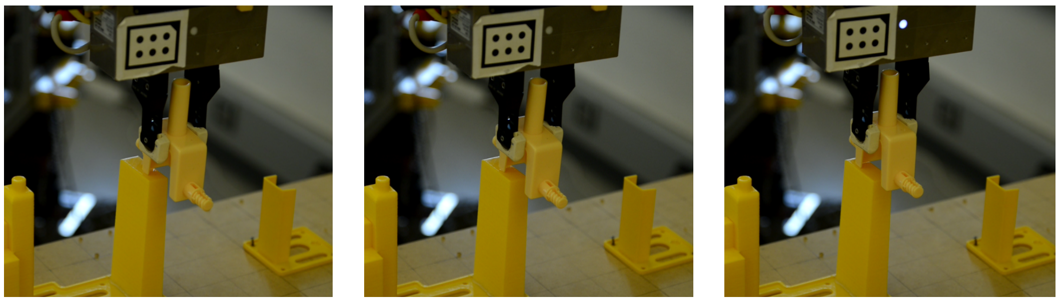


FIGURE 4.4: Starting with a high offset in positive X the gripper gradually adjusts to the correct pose.

evaluated to classify the current offset with the heuristic described in section 3.2.2. Then the gripper retracts. At least four times the gripper pushes down and retracts again, until one offset can be clearly determined over all results. If this is the case, the gripper adjusts its position along X and Y at 0.1 meters along one axis. Eventually the actuated object lands on the target without contact (top-middle image in 4.3).

This push, retract, evaluate, adjust cycle repeats until no contact forces are detected. In the end, when the rim is around the box, it succeeds, if not it fails. The difference is determined by measuring the distance between hollow box and bottom cube.

Results: 98 out of 100 random scenarios were successful. The only two failures were in such cases where the target is placed at [-0.07, 1.19] and [1.16, -0.02], where the actuated object and target completely miss each other when pushing down. The bottom plot in figure 4.3 shows the target's offset in X and Y, in relation to the amount of contacts needed until successful assembly. In table 4.1 the resulting data is listed. All 98 successful runs together took 5236 seconds (1h 17m 16s), the two failed runs immediately terminated within a second. Considering a success rate of 98% the procedure for end effector adjustment achieves its goal and H_0 is disproven.

4.1.2 Real-World Application

Contrary to most Peg-in-Hole tasks, assembling the Battat airplane from the YCB dataset includes manipulation of multiple differently shaped objects, therefore each step of the assembly must be addressed separately. Grasping different shapes is done with a designated orientation of the gripper per airplane part. Also, each two parts require a specific orientation to be put together. Implicitly, this changes the wrench-sensor's orientation as well. Therefore the heuristic must be adjusted, depending on the gripper's orientation during assembly, because the expected torques

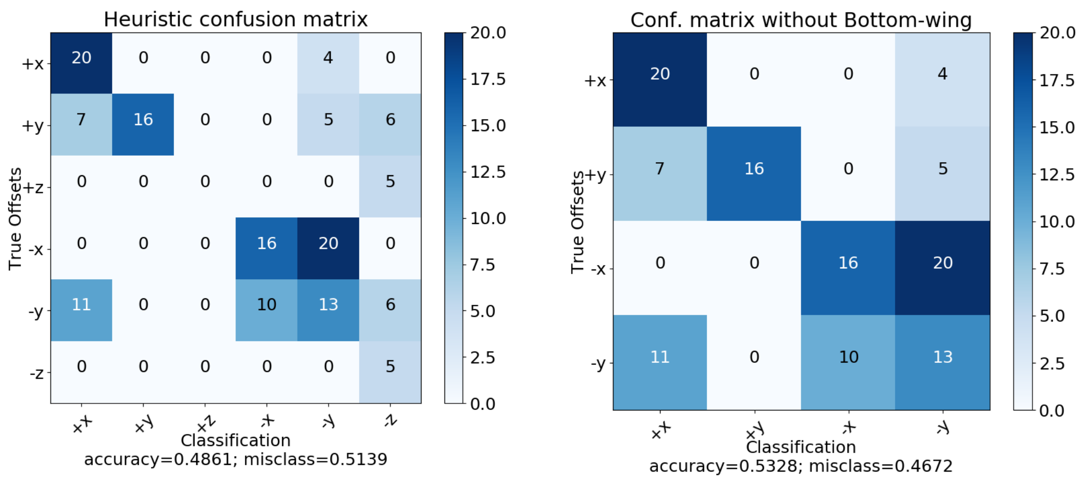


FIGURE 4.5: Left: Confusion matrix over all observed data sequences. Right: Same, but without bottom-wing.

during contact events change w.r.t. the wrench-sensor's orientation and contact points between the two parts to assemble. Nevertheless, it only classifies an offset in four directions by comparing the torque's strength on two axes.

Five steps of the assembly and their contact forces are observed, namely they are (1) chassis on holder, (2) bottom-wing on chassis, (3) underbody on bottom-wing, (4) upper-body on underbody and (5) bolt through rear bodies. To test the heuristic for each of these scenarios, first the actuated object, held by the gripper, is moved into successful assembly with the target. From this goal position a horizontal offset of 5 to 20 millimeters is applied, which resembles the current inaccuracy of the robot, determined later in section 4.2.2. Each of the four directions are tested separately. The gripper is moved downwards until the actuated and target object are in contact.

For each scenario and offset, wrench data of at least five contact events are recorded. Afterwards, the recorded data is classified by the heuristic in section 3.2.2. The results are illustrated as confusion matrices between the actual offset in real-world and the heuristic's classification during contact. See appendix A for an evaluation of each scenario and offset. The results in this section show the general performance of the heuristic.

"The heuristic is at least 80% accurate when classifying the gripper's offset based on contact forces" is the hypothesis to be answered here, so the H_0 hypothesis to be disproven is, that the heuristic is not accurate enough to confidently classify the gripper's offset.

Results: As shown in figure 4.5, the heuristic's accuracy over all recorded force-data is only 0.4861%. When leaving out the scenario of putting the bottom-wing onto the chassis, accuracy rises to 0.5328%. A summary of each scenario's accuracy is shown in table 4.2. This heuristic is clearly too simple to reliably classify the gripper's offset based on real-world contact forces, but at least it is better than guessing (0.25%). Also the amount of data is barely enough to calculate any significant improvement (e.g. from student's T-test). The H_0 hypothesis can't be disproven.

While training a proper classifier on accuracy was not the intend of this thesis it is still interesting to see the difficulties of evaluating contact forces between

TABLE 4.2: Accuracy rating of the heuristic per scenario.

Scenario	(1)	(2)	(3)	(4)	(5)	Total
Accuracy	0.2609	0.2273	0.7500	0.3871	0.6591	0.4861

every-day shaped objects. As already mentioned by (Bouchard et al., 2015) friction in rigid-rigid contact is a big problem, because the axis of applied force changes due to the object's stress, which influences measured contact forces. Another factor is the arm's joint-impedance controller, simulating a spring-like behavior. When the gripper pushes against the target, the arm's joints are not rigid enough to hold the gripper in position, but instead give like springs, pressing the gripper in a direction of relaxing the joint's stress. Even with higher impedance per joint this issue occurs.

Chassis The influence of friction and the arm's flexibility is most prominent in the first assembly step (top-left image in figure 4.2). Pushing the chassis down against the holder invokes tension in the arm, which is relaxed by pushing the chassis away form the robot, towards positive X. This causes strong torques around T_Y which confuses the heuristic: offsets in +/-Y are classified as +X.

Bottom-wing Assembling the bottom wing is a good example for why the gripper's position is important to produce informative wrench data in contact (middle-left image in figure 4.2). With a gripper oriented horizontally, the contact forces are very similar over all offsets. Unfortunately, grasping the bottom-wing at a different angle was not possible, because the wing's dimensions were bigger than the gripper's maximum opening.

Under- and Upperbody A huge difficulty surely is the diversity and asymmetry of the shapes. For assembling the upper-body onto the underbody (left images in figure 4.2) an offset in right and left (+X, -X) can be easily determined, while this is much more difficult along the Y axis because there are no distinct contact points between the two object's shapes.

Bolt Putting a bolt into the rear-hole was the most difficult task, mainly because of the upper-body's surface around the hole (bottom image in figure 4.2). This scenario has been tested with several different offsets, yielding a variety of torque forces, especially along -X direction. At small offsets the bolt landed near the hole, pressing against the inner rim. At bigger offsets and oscillation overshoots the bolt glides off the upper-body's side instead, yielding completely opposing torque forces. In +Y direction the bolt glides down the rear wing, while in -Y the bolt presses against the upper-body's middle region.

4.1.3 Summary - End Effector Adjustment

After investigating real-world contact forces and the heuristic's accuracy it is foreseeable how the end effector adjustment procedure from section 3.2 would behave in the real world. Even randomizing pushing force (as described in section 3.2.3) to cause variety in the torques, and taking the most frequenting classification, doesn't help the classifier's accuracy when putting the chassis onto the horizontal holder. Nevertheless, the Gazebo simulator shows, that a better classifier would make an application in the real world promising. Force feedback has been successfully implemented in the cognitive plans of CRAM, contributing to its reactionary mechanism during intricate, force-sensitive assembly tasks.

4.2 Object Pose Adjustment

To evaluate the procedure of object pose adjustment from section 3.3 the gripper's accuracy is measured before and after touching objects in the real world. The improvement depends on estimating contact points through pseudo ray casting in simulation (see section 3.3.1), and detecting actual contact points in real-world (section 3.3.2). Only if both are accurate, precision can be improved. Both components are taken into account separately.

4.2.1 Simulation in Bullet-Physics - Pseudo Ray Casting

The estimated contact point between two rigid bodies is obtained in Bullet by pseudo ray casting, explained in section 3.3.1. Their accuracy is mostly dependent on Bullet's contact point calculation, which again is heavily influenced by each object's grade of detail in its corresponding mesh representation. As already shown in figure 3.6 during explanation of the procedure, the estimated contact point do not lay exactly on an object's surface. This is because the collision mesh of each object is slightly bigger than its visual shape.

However, the precision of Bullet's contact points can be investigated, based on the meshes used. Provided that all included meshes are decomposed into a composition of convex hull shapes (see pseudo ray casting in section 3.3.1), two objects are in contact when their actual shapes overlap, not their bounding boxes, which is the usual case if the mesh is not decomposed.

To obtain how far Bullet's calculated contact point is away from the actual surface, the calculated point is retrieved and visualized first. Then a linear translation is applied to the point until it is visualized directly on the object's surface. This error gives the distance between calculated and estimated contact point.

When pseudo ray casting the chassis against the holder on eight different sides, Bullet calculates the contact point 1.2 to 1.7mm off the holder's surface. Vice versa the contact point on the chassis' axis is approx. 1.5mm before its axis' surface. The bottom-wing's contact points are similarly distant (1.0 - 1.4mm) to its actual surface.

Changing the step distance for pseudo ray casting has no impact on the retrieved results. Overall the contact point offset lies between 1.0 and 1.7mm, depending on the mesh and its surface. This offset is taken into consideration when adjusting the object's pose on touch, but since it varies, some error persists.

4.2.2 Real-World Application

The `touch` procedure from section 3.3.2 is evaluated through real-world application. The tests include (1) touching the MDF board without an object held, from top, front and left, before picking up the chassis, then (2) touching the horizontal holder with a chassis held, from front and left, before placing the chassis onto it, then (3) touching the bottom-wing without any object held from left and front, and (4) the chassis on the holder from left and the front. Afterwards (5) the bottom-wing is picked up and placed onto the chassis.

To investigate the increase in accuracy, first the real-world error must be determined. Therefore all object's positions in Bullet and real-world are synchronized

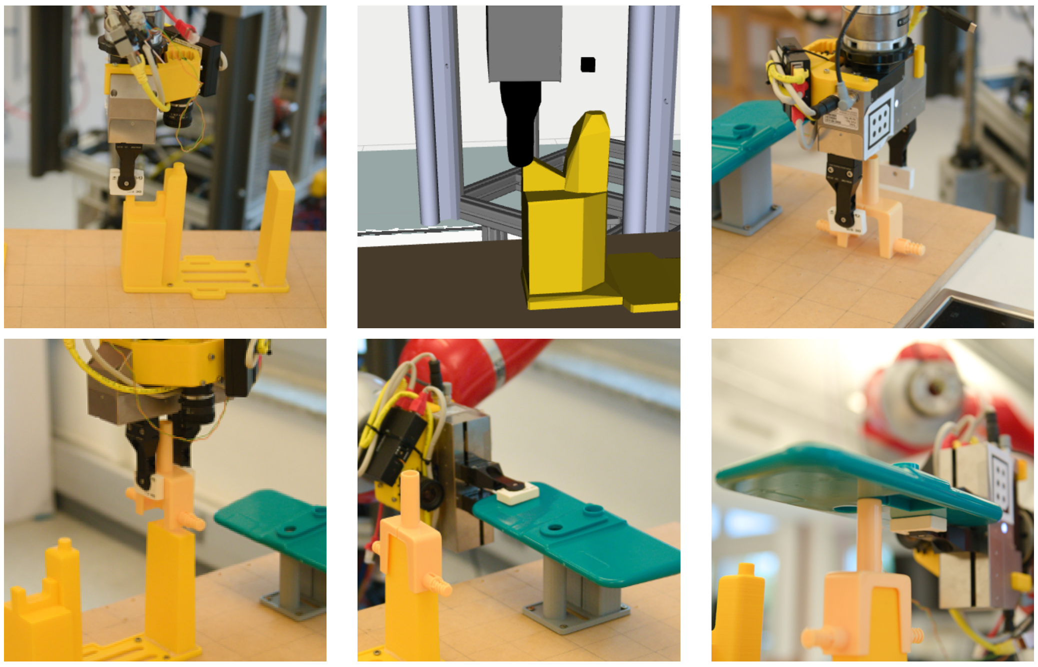


FIGURE 4.6: Real-World inaccuracy without object pose adjustment. (Top left) Holder's position in the real world as reference, (top middle) calibration in Bullet (top right) (1) grasping chassis, (bottom left) (2) putting chassis onto pedestal (bottom middle) (3a) grasp bottom-wing (bottom right) (3b) putting bottom-wing onto chassis.

first. This is done by moving the gripper-tip in real-world to a distinct position of an object, to then cast the object in Bullet until the simulated gripper and object are aligned in the same way.

To test the real-world error, the robot is moved away from the assembly board by teleoperation. Then the torso, as well as the arm joints are arranged (homed) into a different joint-configuration. Now the robot moves back to the assembly board. When the gripper is moved to one of the distinct poses, used to synchronize an object, the position differs. The real-world inaccuracy without object pose adjustment is depicted in figure 4.6 and table 4.3.

Three tests show how accuracy is improved by touching objects: (1) touch the board to pick the chassis, (2) touch the holder to put the chassis on it and (3) touch the bottom wing and chassis, before picking up the wing (3a) and putting it onto the chassis (3b). Each object's position has been adjusted before a tests begin, and the robot is retracted from the assembly board and its arms configured in a homing position, before each test is executed. To verify an object's position it is touched from two perpendicular sides. After an object's position is verified, the following action is analyzed regarding its accuracy, which means in test (1) for example, how far off the chassis is grasped from the desired position. The hypothesis to prove is, that object pose adjustment improves accuracy in the investigated assembly tasks, therefore the H_0 hypothesis to disprove is, that there is no improvement.

Results: As depicted in figure 4.7 grasping the chassis (1) was still done with a slight offset. With an incorrectly held chassis, touching the horizontal holder (2) inherits the error, which is malicious along the Y-axis but beneficial in X, because the

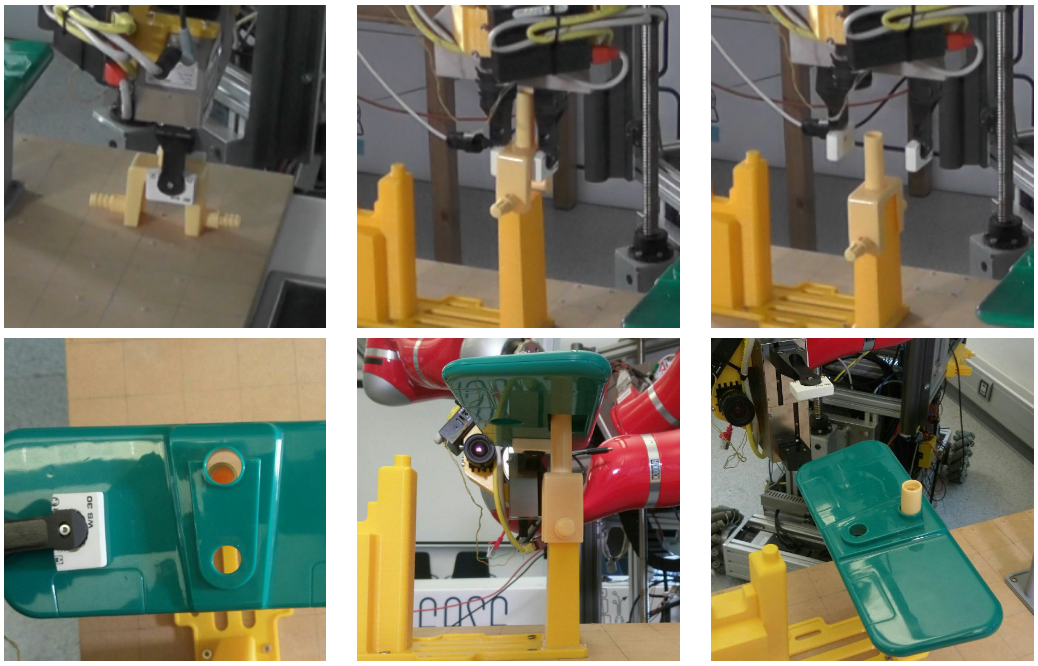


FIGURE 4.7: Manipulation accuracy after Object Pose Adjustment. (Top-left) Grasping the chassis after touching the board, (top-middle) holding the chassis above the holder's pedestal, (top-right) dropping the chassis onto the pedestal. (Bottom-left) Offset when putting the bottom-wing onto the chassis from top perspective (bottom-middle) same from the side, (bottom-right) successful bottom-wing on chassis assembly.

TABLE 4.3: Real-world X, Y offsets in millimeters with and without object pose adjustment.

Scenario	**(1)**	**(2)**	**(3a)**	**(3b)**
W/o pose adjustment	26-32, 8-12	15-20, 6-8	28-33, 1-4	22-27, 12-18
With pose adjustment	0-3, 2-5	0-3, 1-4	2-4, 1-3	0-2, 2-7

error in the chassis' translation to the gripper can be omitted, as shown in the top middle figure in 4.7, where the chassis is only slightly off to the right.

After touching the bottom-wing and chassis (3) the former was picked up to put on the chassis. Still with a slight offset (bottom left and middle image in figure 4.7), adjusting the parameters for contact point calculation for the bottom-wing corrected its offset, to finally assemble the bottom wing onto the chassis (bottom right image).

By keeping the base still after touching an object, differences in localization can be omitted and since the torso's joint only affects the gripper's vertical position (along robot's Z axis) any error in the prismatic torso-joint can be ignored. Unfortunately, inaccuracies in the arm's joint-states persist, which adds to the error in determining the estimated contact point in Bullet (see section 4.2.1). When pressing the gripper against an object, the arm's joints give, because the joint-impedance controller is designed to simulate a spring-like behavior. To what extent this error influences object pose adjustment, see section 4.3.1.

Over several test in the real world the offset for contact-points, the gripper's

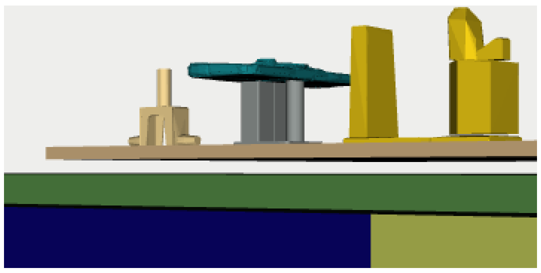


FIGURE 4.8: Lifted assembly board to mitigate the object's error in its vertical position, based on the gripper's contact pose.

fingertip size and mitigation of the arm's flexibility were adjusted to get a successful execution of all assembly steps separately. After all parameter-adjustment and several object's touched a small offset was still present, depending on the assembly step. Compared to the offsets without object pose adjustment, the improvement is a strong enough indicator for the procedure to be applicable in real world.

4.3 Working with Robots in the real world

The transition of robot's activities from simulation to real-world carries several difficulties, which are elaborated in the following section.

Boxy is a home-brew robot, designed and assembled in the Institute for Artificial Intelligence. Compared to other robots, all technical support is provided by the staff. In general the robot works fine, but some of the software and hardware components have potential for improvement, which is elaborated in this section. Some improvements were partially contributed to in the development of this thesis, by extending the robot's drivers and equipping Boxy's system with new frameworks like Giskard.

4.3.1 KUKA LWR-4+ Arms

In freshly manufactured LWR-4+ arms the differences between actual and simulated joint state, retrieved from each optical encoder, are only around 1-2% (Hirzinger et al., 2002) but this error can accumulate over all 7 of the arm's joints down to the gripper. Also the gripper's weight and inertia must be calibrated accurately to keep its pose consistent. To what extent this impacts the assembly tasks in general is hard to determine, but an observation was made (see figure 4.8) where the real-world gripper hangs lower than the simulated joint-states imply. Therefore the whole assembly board is lifted by 2cm in Bullet, to mitigate the gripper's vertical offset. Zeroing the arm's joints was done manually, which may have contributed to their inaccuracy. Measuring the distance between the arm's base in simulation and real-world yielded no significant difference, therefore it is assumed that the error originates from the arm's calibration and hardware's wear.

Adjusting the joint's impedance is a matter of calibration, in which different setups have their pro's and con's. Higher impedance make the arm stiffer, which increases contact forces, while less impedance enables some joints to damp impacting stress through contact forces at the right rate. Stiffness can only be achieved to a certain degree, where the joints still damp incoming forces a bit, even with very high impedance (double the below mentioned numbers). Lower impedance on joints close to the gripper makes it possible to press with the gripper against an obstacle, while the force is distributed along the joints set to lower impedance, allowing to omit some of the force from the gripper in contact. This kind of spring-like elasticity and damping in arm joints does not influence their simulated joint states, since all joint positions are determined through optical encoders within them. Low impedance overall on the other hand, can still decrease accuracy, and increases the above mentioned vertical offset. The impedance used over all joints is [500, 500, 500, 400, 300, 200] N/m, which works stable enough for all use-cases, without damaging the plastic objects during collision.

The arm's joint controller, below DLR's Links-And-Nodes interface, died unexpectedly a couple of times, leaving all joints at zero impedance.

Because of how the arm is attached to the torso, its configuration space is very restricting when planning joint trajectories in Giskard. Even though each joint has a pretty wide limit range (+/-170°, +/-120°, +/-170°, +/-120°, +/-170°, +80°/-45°, +60°/-30°)[2] Giskard often ran into difficulties finding a joint trajectory. Compared to Willow Garage's PR2 this issue was rare, because its arms partially use infinite rotational joints. The work of (Chaves-Arbaiza, García-Vaglio, and Ruiz-Ugalde, 2018) could have helped finding a suitable configuration of mounting the arm to Boxy's torso. It simulates several configurations, evaluating each by its ability to grasp objects. Also (García-Vaglio, 2020) can be considered for this, which rapidly determines robot capabilities based on GPU calculation.

4.3.2 Localization

Usually Boxy has two Hokuyo LIDAR in its base. Unfortunately, the front-left sensor doesn't work, such that Boxy only receives its laser data at 270 degree to its back-right side, which was thought to be enough for localization at millimeters of precision. Later, however, the offset was found to be around 2-3cm between real-world and simulation. This makes localization way harder, which hindered as well as inspired the thesis' contribution. Using only the hind laser sensor Boxy's localization was always a bit off, which can be seen for example in figure 3.5, where the MDF plate is moved off the table in simulation, while its edge should be aligned with the table instead.

Over several months of testing in the real world, Boxy has been moved by hand in between session. The LIDAR must have taken a hit, whereby it was rotated ever so slightly, causing the base's localization to be off by only a few millimeters. Accumulating this error up to the gripper it had a huge impact of several centimeters on the gripper's pose, not to mention a rotational offset. Even now the functioning LIDAR's orientation is prone to be off, because its holder is susceptible to be bent.

4.3.3 Giskard

Controlling robots is hard. Giskard has been thoroughly explained in chapter 2.4. Every time that Giskard fails to generate a joint trajectory, it is due to the arm's configuration space, as explained in section 4.3.1. Therefore the arm had to be brought into a promising configuration first, before giving Giskard the task to plan a trajectory, which increases execution time of the assembly steps a lot. In figure 4.9 these joint configurations are depicted.

Boxy's torso controller was included into Giskard's trajectory planner during development on this thesis. After the base's joint controller was included as well, Giskard was able to plan trajectories for the gripper's goal pose, even if they were

[2] https://www.dlr.de/rm/en/desktopdefault.aspx/tabid-12464/21732_read-49777/

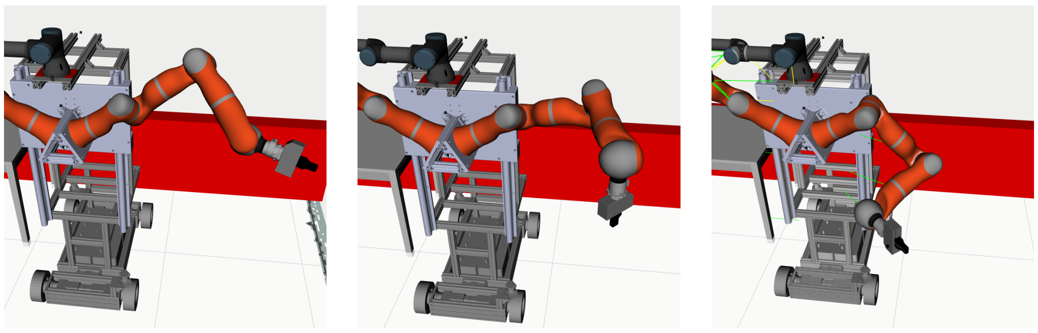


FIGURE 4.9: Arm standard configurations to start planning motion from. (Left) before any cartesian goal for the gripper, (middle) for top grasping and touching objects (right) picking and placing the *bottom-wing* and *top-wing*.

out of Boxy's reach, utilizing not only the arm and torso, but also the base. To do this, Giskard is giving each involved controller joint velocities over time. In simulation these trajectories are executed without a problem, but for real-world application each controller has to be synchronized, which is currently not the case. Especially the recently included torso-controller executed its joint velocities way before the arm's controller, while the arm was taking approximately 1.5 times longer than Giskard predicted. This is a huge issue when planning a trajectory with collision avoidance. Therefore the arm is brought into a safe configuration every time a potential collision is expected or when the gripper's target orientation strongly differs from its current one.

A controller's task is finished, when their affected joint velocities are below a certain threshold, at least this is how Giskard determines if a controller has terminated his process. Giving the arm-controller a joint-trajectory does not result in a completely continuous movement, but usually pauses somewhere in the middle of the trajectory before finishing the latter half. Since CRAM can only know if a movement is done by listening to Giskard, which signalizes termination before the arm is actually done moving, CRAM is currently not capable to move the gripper to several different poses when working with Boxy. Therefore the procedures in section 3 were paused after Giskard's response until the movement was actually done.

4.3.4 KMS40 Force-Torque Sensor

Communication from and to the KMS40 sensor is done via Telnet, wrapped within a C++ ROS driver that propagates the force data into the ROS network. Through this wrapper it is possible to receive data and send commands to the sensor, e.g. putting the current force offsets to zero. Receiving data and publishing it into ROS' network is stable, but sending commands to the sensor is not.

The sensor constantly sends its data as ASCII strings to the C++ driver. When the driver wants to send a command to the sensor it expects a specific response, verifying that the command was accepted by the sensor, but when the sensor gets hotter its performance decreases, which manifests in stuttering data transfer. If the driver sends commands to the stressed force sensor, it does not respond immediately, instead it takes a while to execute the given command while still sending force data, which is not the response expected by the driver, so it terminates.

Since zeroing force data is of elementary importance for reasoning, a solution was found which can work solely on the received force data. By subtracting the currently sensed forces from upcoming data, a wrapper was used that takes care of zeroing the data and providing a corresponding service to do so, instead of communicating directly with the low-level C++ controller or sensor.

The sensor was not explicitly tested for cross-talk between its six axes. Manually applied force to the gripper yielded results as expected. Through the above mentioned nodes, the emitted data was filtered to such a degree, that noise was not an issue.

4.4 Evaluation Summary

Both approaches from chapter 3 have been thoroughly tested in an assembly environment, manipulating objects by an autonomous robot. The first approach seems stable in simulation with a high rate of success (see table 4.1), verifying the procedure of Gripper Pose Adjustment on the real robot, however, the classifier is not accurate enough to apply this procedure in the real world (see table 4.2), which is visualized by several confusion matrices per assembly scenario in appendix A. On the other hand, adjusting the airplane parts by tactile perception before manipulating them increased the robot's accuracy significantly (see table 4.3), which made it possible to assemble the chassis onto the holder and bottom-wing onto the chassis. High offsets are not feasibly accountable, which is explained in section 5.2.

Considering controller issues, hardware inaccuracy and offsets in localization, the gain in precision is immense, mainly because issues in the localization were omitted by keeping the base still between touching and manipulating an object. Changing the LWR-4 arm's joint-states yielded stronger offsets of the gripper between two configurations than expected, which became high when the bottom-wing is picked from a completely different joint configuration than it was touched before. This resulted in high offsets when picking it up.

In summary: with a better classifier the first approach could be stable enough to be combined with the second approach, however, this statement can't be verified without working examples. Touching objects can adjust an object's position well enough to pick and place them confidently, while adjusting the gripper's pose during rigid-rigid contact between the actuated and target object could potentially help mitigating remaining inaccuracy.

Chapter 5

Conclusion

5.1 Summary

Industrial assembly tasks evolve from a static sequence of movement to autonomous performance. The two approaches presented have shown, that autonomous robots are capable of performing assembly tasks, even if the robot is not rigidly attached nearby an assembly line, but can move around freely. Using a cognitive environment is an important foundation to react dynamically, and gaming engine physics are helpful to improve the quality of a belief state. Force feedback is a great benefit in this domain, providing important information at run time to utilize cognitive procedures in object manipulation the right way.

When looking at rigid-rigid contact forces, research has progressed far, but handling a variety of shapes reliably is still not quite possible, at least it is not as simply achieved as with the presented heuristic. The concept of adjusting a gripper pose offset based on contact forces, however, seems to be a promising endeavor in the right direction to improve an assembly-robot's capability to achieve its goal in uncertainty.

To handle uncertain positions of assembly parts, touching each object helps to reduce this uncertainty, as shown in the second approach. This again was only possible because of a sophisticated belief state like a gamin engine, which is capable of calculating certain features like contact points between simulated objects in three dimensional space. By combining reasoning on the actual and simulated environment in contact events, the inaccuracy of an object's believed position can be decreased by orders of magnitude.

5.2 Discussion

Finding a proper classifier is hard for Peg-In-Hole tasks already and doesn't get easier with a diversity of shapes. However, most every-day scenarios usually require precision in the order of millimeters not micrometers, as opposed to industrial endeavors. As mentioned in related work of section 1.3 the work of (Stelter, Bartels, and Beetz, 2018) was considered to be integrated as classifier under supervised training. When the execution of assembly tasks transitioned from simulation to real-world, controller problems arose where immediate interruption of continuous movement was programmatically impossible, which made time-series related classification infeasible.

The autonomous robot used in this thesis was approximately 2-3cm off from the desired goal, before applying the mechanism presented in this thesis. Reducing the error to a few millimeters was enough to successfully perform an assembly, because the airplane part's holes and pins generously allow errors of this scale. For both approaches a maximum offset of 3-4cm was assumed. Any inaccuracy higher than that is not proficiently solvable, which is mostly because of the object's size, for example touching the holder allows an offset range of about 3.5cm from the estimated contact point. Trying to touch, perform or assemble an object under higher offsets would probably miss the target or yield unreliable data.

The subset of airplane parts to assemble is selected with purpose, because each assembly involving these parts is done by moving downwards. This choice was made primarily for the first approach, where contact forces are evaluated to classify the gripper's error from the actual goal position, but also because of the arm's enormous wrist. These assembly scenarios can be executed with the gripper pointing down, since the arm requires around 40cm of free space between the gripper's fingertip and the wrist. A horizontally oriented gripper close to the assembly board has been causing a lot of potential collisions in simulation. An exception to this is the bottom-wing assembly onto the chassis, where the wrist and gripper are still quite low, but off the assembly-board's range.

Touching objects allows to adjust their believed position on the X and Y plane. During the beginning of development for this procedure the robot's localization was off around its Z rotation, causing massive offsets between real-world and simulation for the gripper. The issue was caused by a small rotational error in the hardware, where the LIDAR was shifted by only a fraction of a degree. When adjusting objects by touch from different sides, the adjustment was completely off because of this. After pushing the LIDAR back into correct position, the procedure became stable. The gist of it: rotational errors in localization are not accounted for in the second approach.

5.3 Recommendations on Future Work

When using autonomous, humanoid robots for assembly tasks the presented approaches can be applied to increase precision. Working with wrench sensors in wrists also requires the right hardware and controller setup. Instead of a spring-simulating joint-impedance controller for the arm a force controller might be more suitable, because contact forces can be better adjusted and friction counteracted. In any case, friction and slipping should always be considered when trying to classify contact forces in real world.

Fine calibration of every joint is crucial to simulate the robot state appropriately, where small errors can easily influence precision of an end effector. Especially localization can ruin a robot's performance, which relies on correct LIDAR configuration, reliable localization algorithms and an accurate map of the environment. The presented approaches can deal with a lot of imperfection, since they actively aim to mitigate these errors. Rotational error, however, make tactile perception much harder.

Cognitive frameworks, gaming engines and wrench sensors are a strong tool-set for autonomous robots performing assembly tasks in uncertain environments. Contemporary work draws their usage in a promising picture for industrial application.

List of Figures

List of Abbreviations

CRAM	**C**ognitive **R**obot **A**bstract **M**achine
IAI	**I**nstitut of **A**rtificial **I**ntelligence
IK	**I**nverse **K**inematic
Lisp	**Lis**t **P**rocessing Language
ROS	**R**obot **O**perating **S**ystem

Appendix A

Real-World Force Data

Following are the average force-torque responses for 5 tested assembly scenarios and a confusion matrix of classifying the data by the heuristic in section 3.2.2. In each scenario the gripper is put at an offset with respect to the gripper's orientation during the assembly. In some scenarios multiple offsets of different distance are taken, when the resulting contact forces are expected to be significantly different. In general the offset is between 5 and 10 millimeters. At least 5 data-sets are recorded for each offset. The data-set sequences are synchronized towards the point of contact, then the average is calculated. The resulting plot is show below each offset of a scenario.

A.1 Chassis on Horizontal Holder

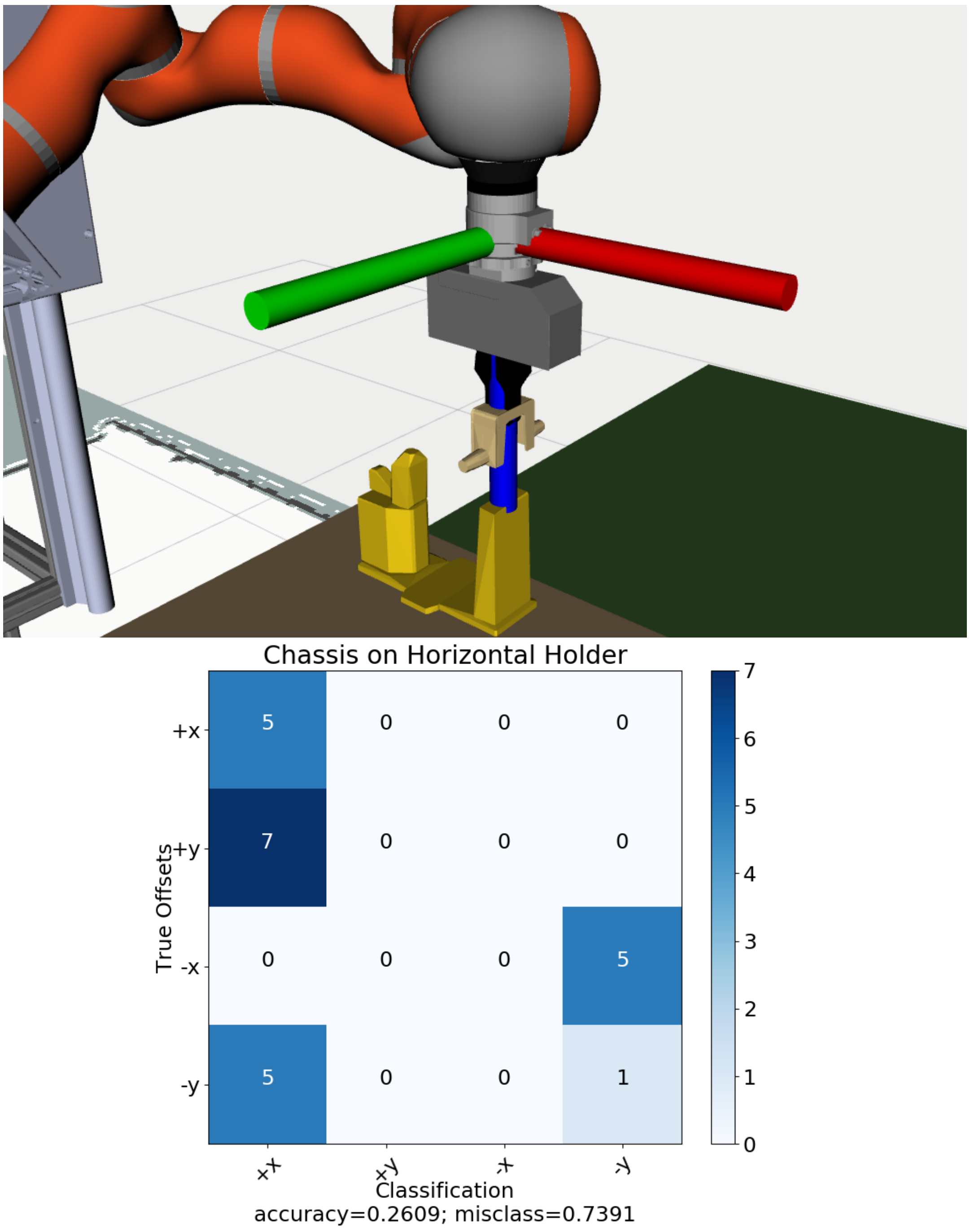
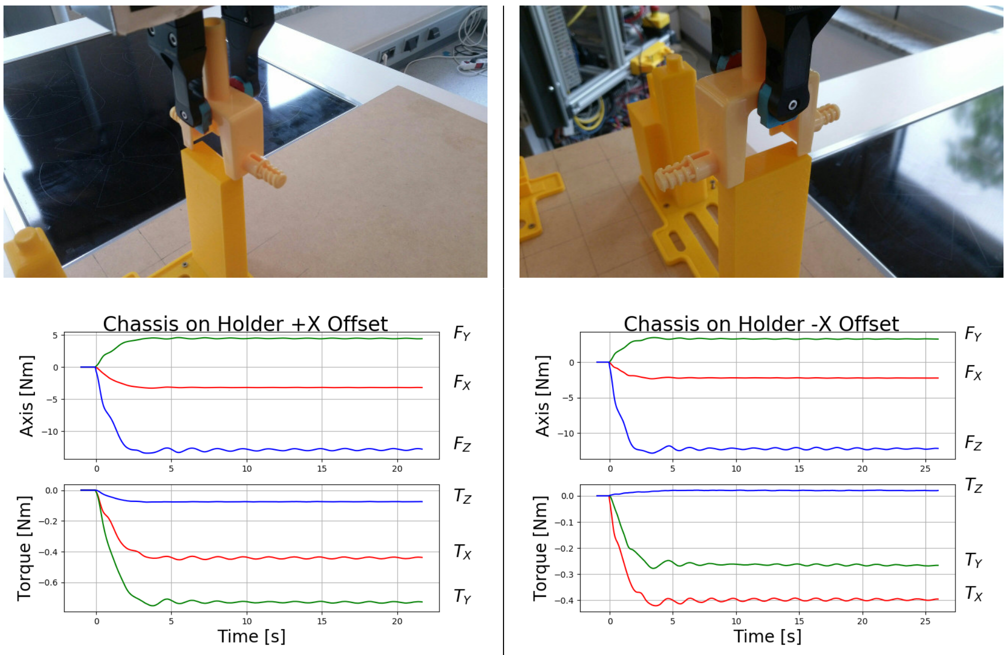


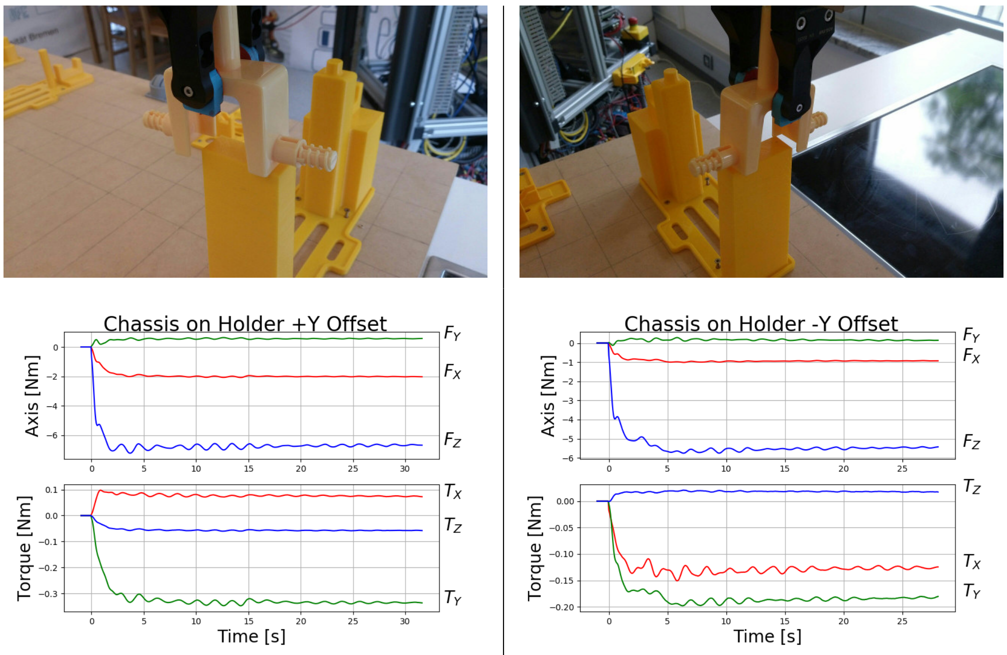
FIGURE A.1: Top: X (red), Y (green), Z (blue) orientation of the end effector while assembling the chassis onto the horizontal holder. Bottom: Heuristic's confusion matrix for this scenario.



+X offset

-X offset

FIGURE A.2: Chassis on horizontal holder forces with X offsets.



+Y offset

-Y offset

FIGURE A.3: Chassis on horizontal holder forces with Y offsets.

A.2 Bottom-wing on Chassis

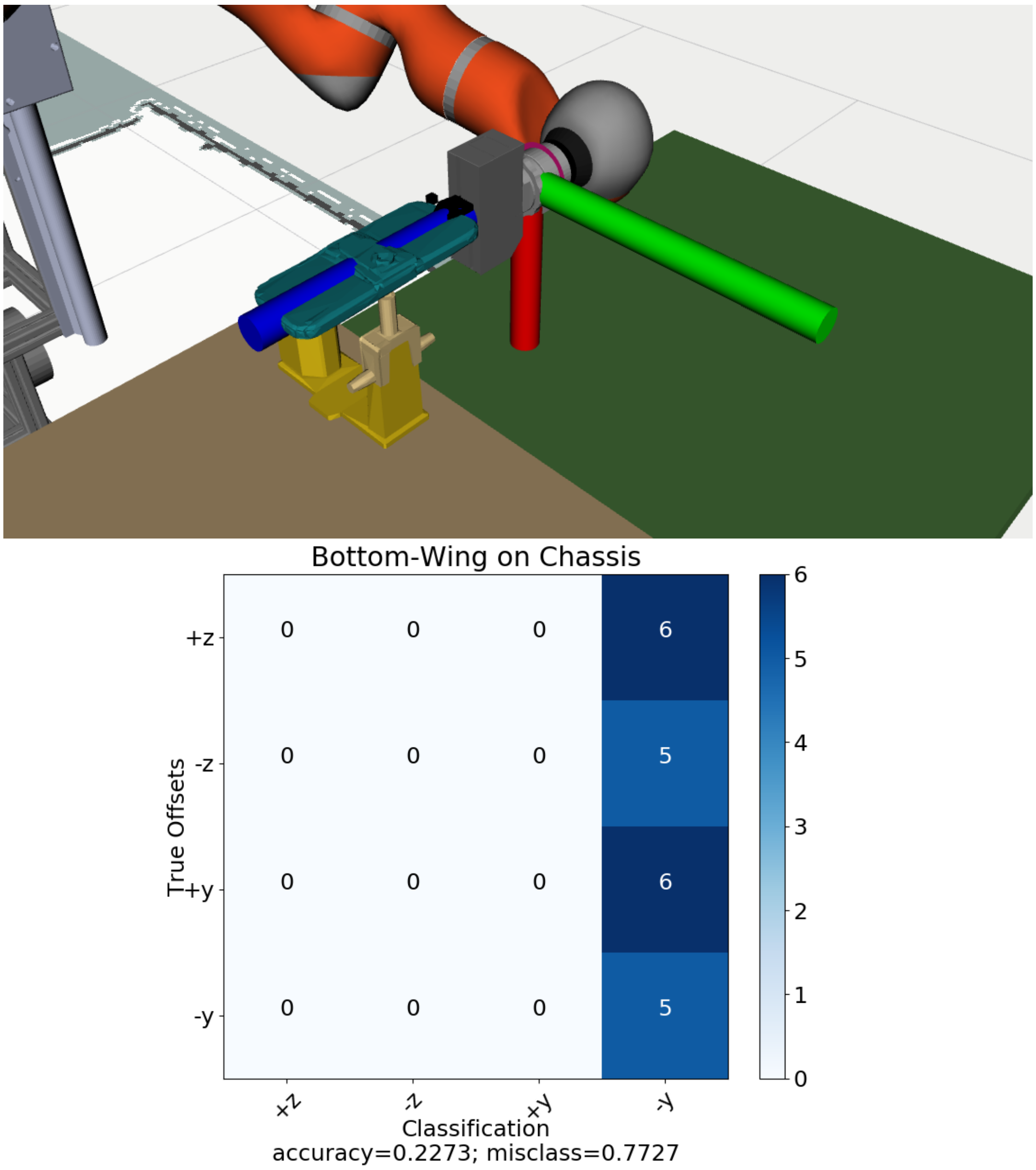
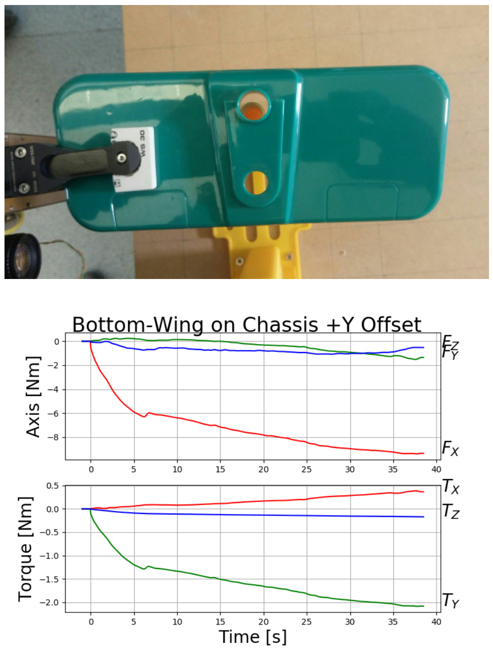
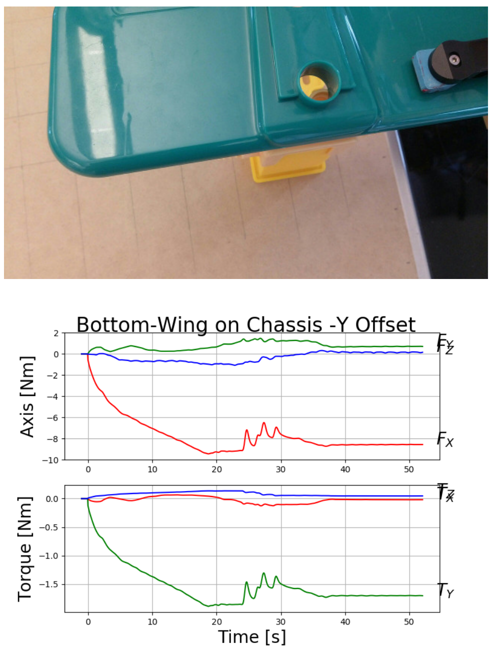


FIGURE A.4: Top: X (red), Y (green), Z (blue) orientation of the end effector while assembling the bottom-wing onto the chassis. Bottom: Heuristic's confusion matrix for this scenario.

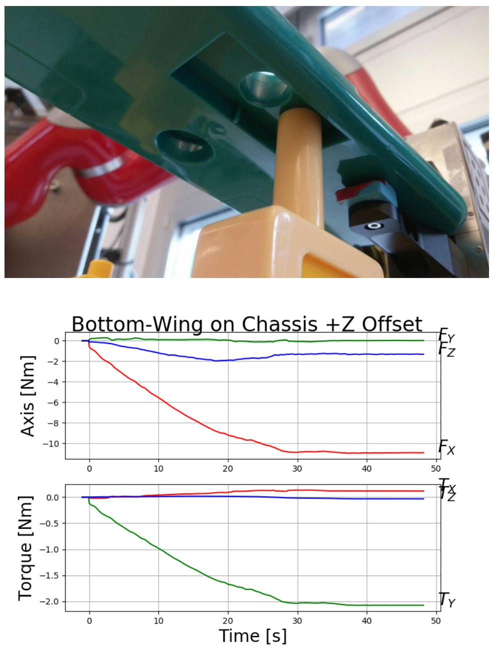


+Y offset

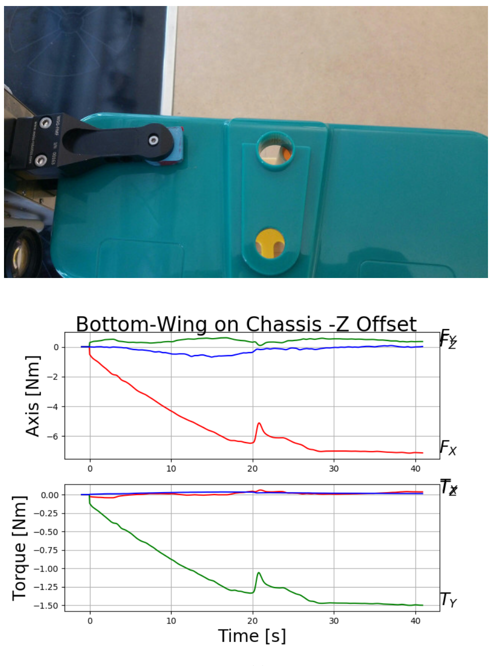


-Y offset

FIGURE A.5: Bottom-wing on chassis forces with Y offsets.



+Z offset



-Z offset

FIGURE A.6: Bottom-wing on chassis forces with Z offsets.

A.3 Underbody on Bottom-wing

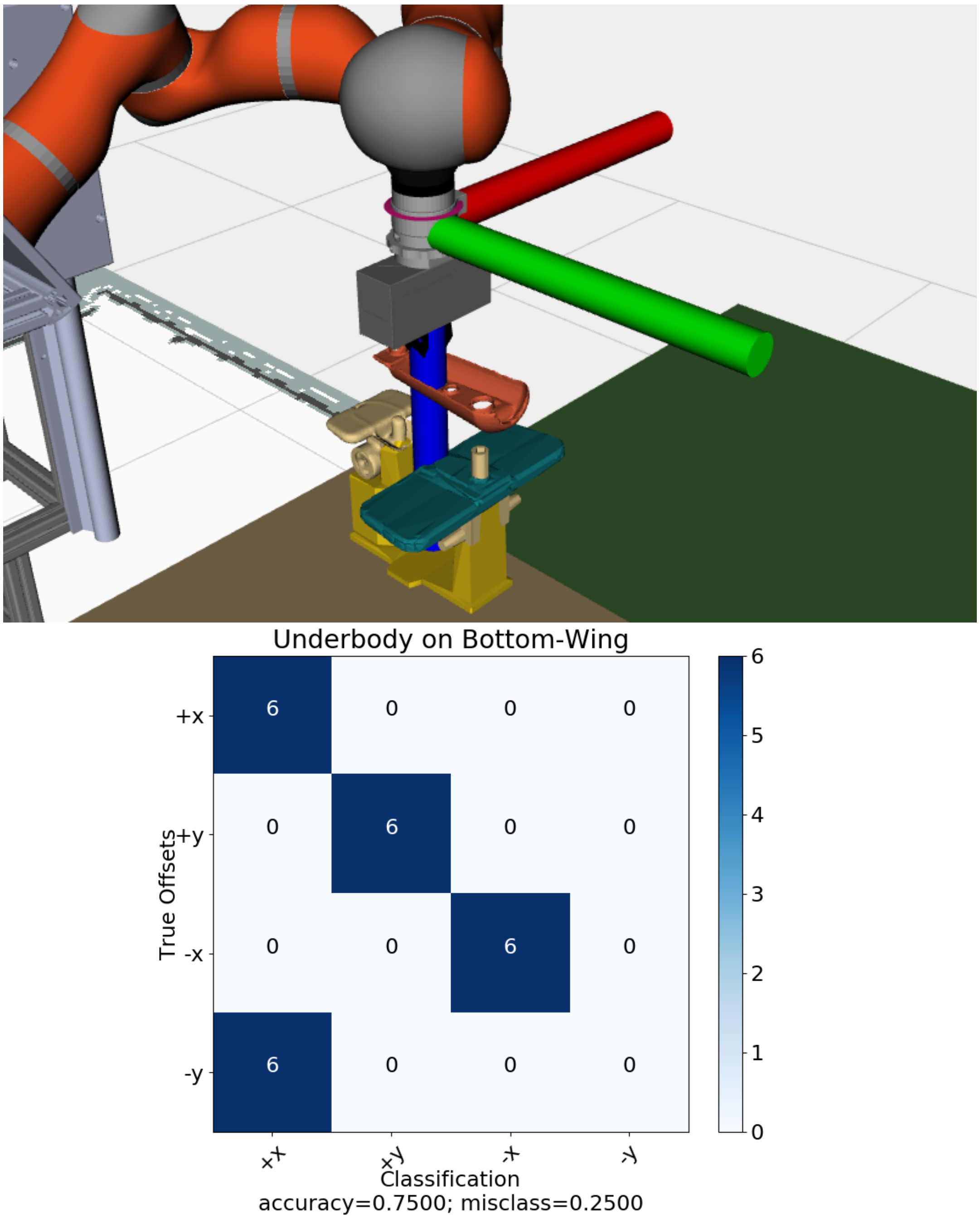


FIGURE A.7: Top: X (red), Y (green), Z (blue) orientation of the end effector while assembling the underbody onto the bottom-wing and rear-wing. Bottom: Heuristic's confusion matrix for this scenario.

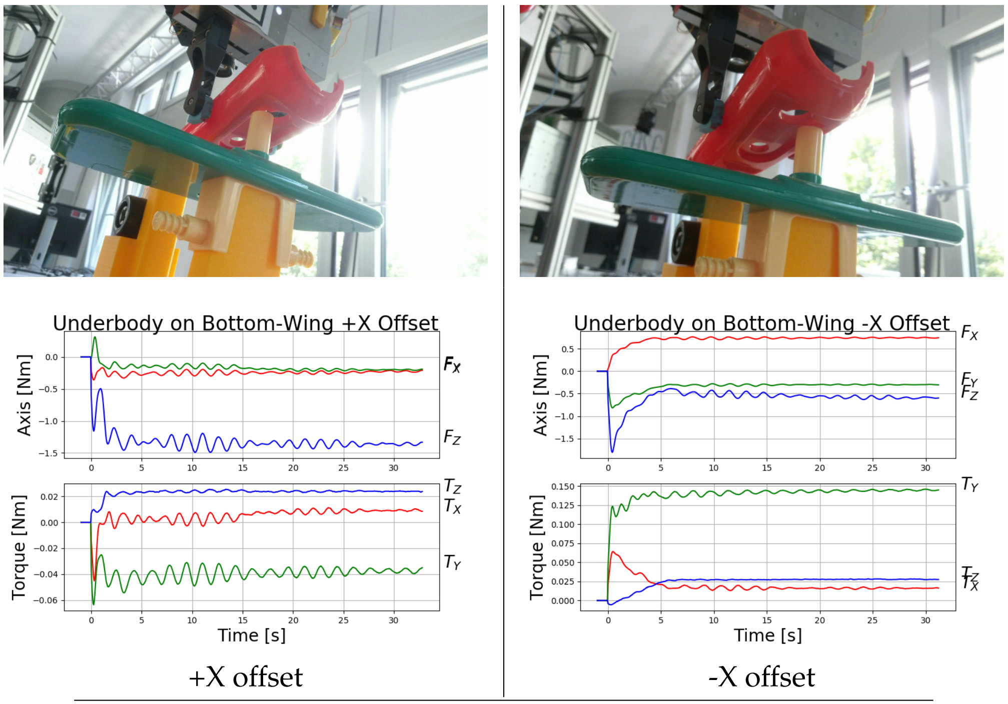


FIGURE A.8: Underbody on bottom-wing and rear-wing forces with X offsets

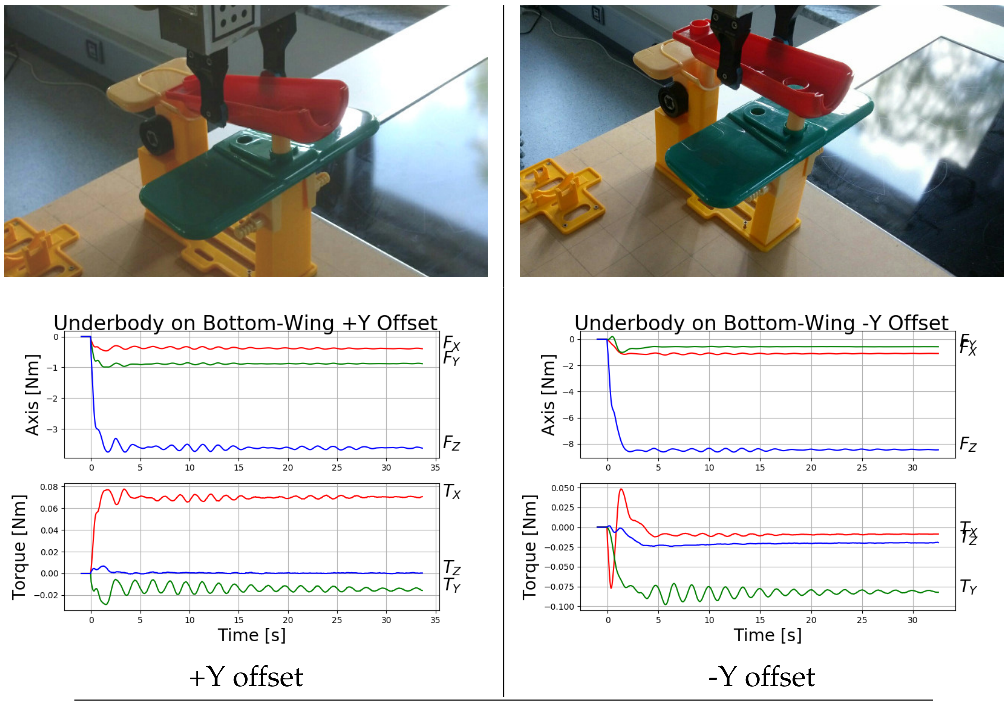


FIGURE A.9: Underbody on bottom-wing and rear-wing forces with Y offsets.

A.4 Upper-body on Underbody

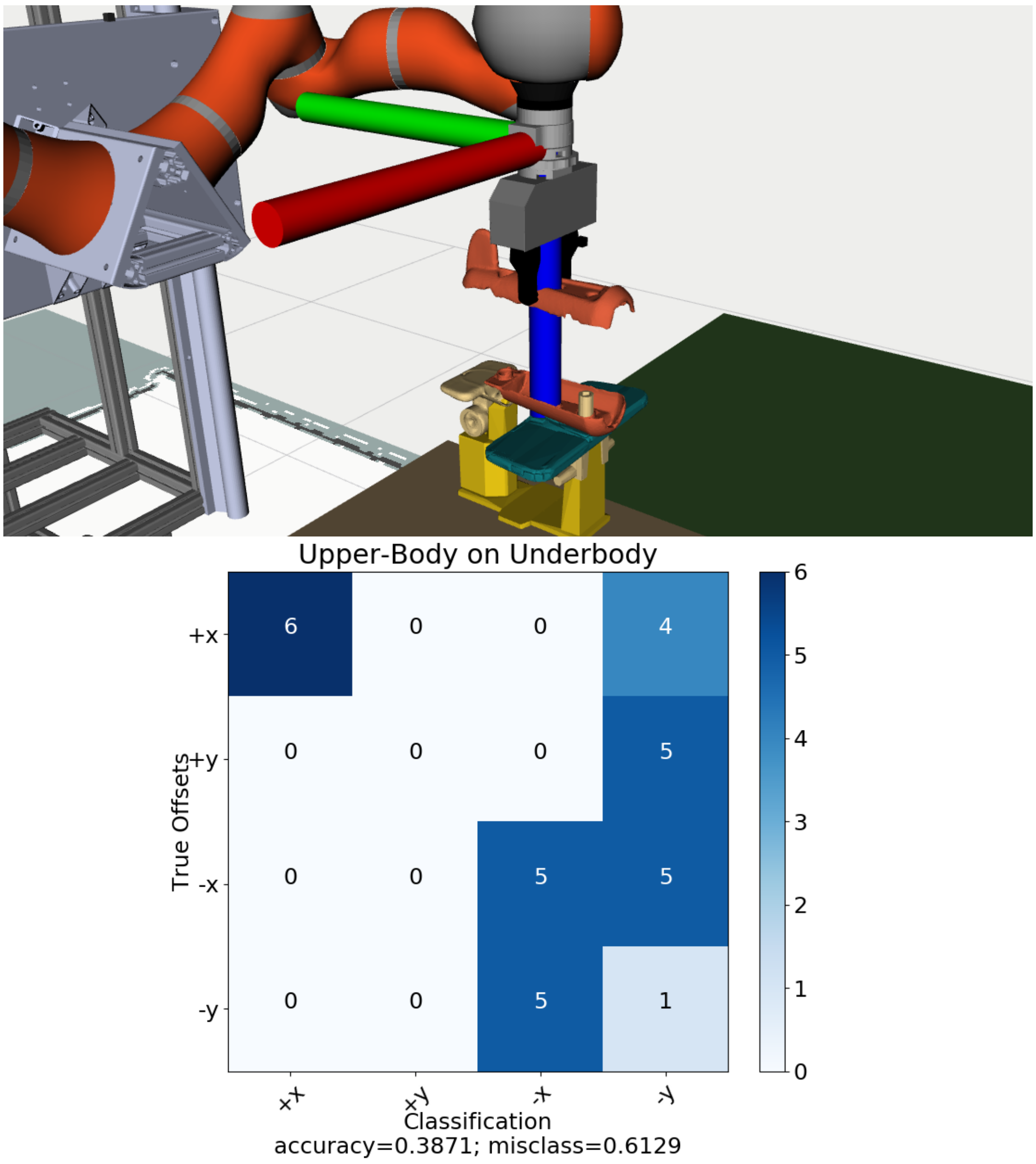
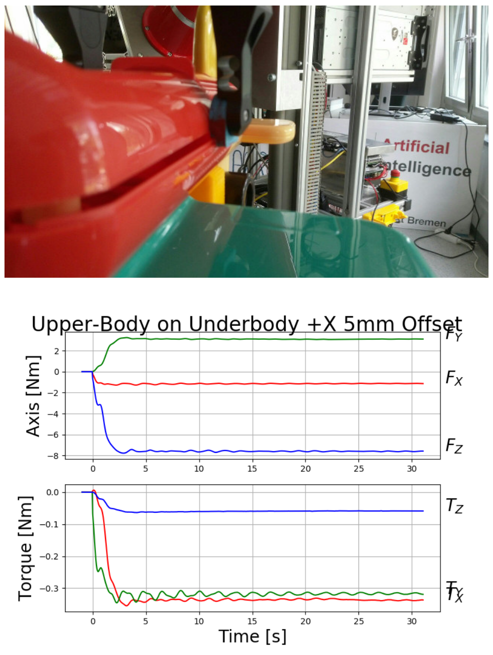
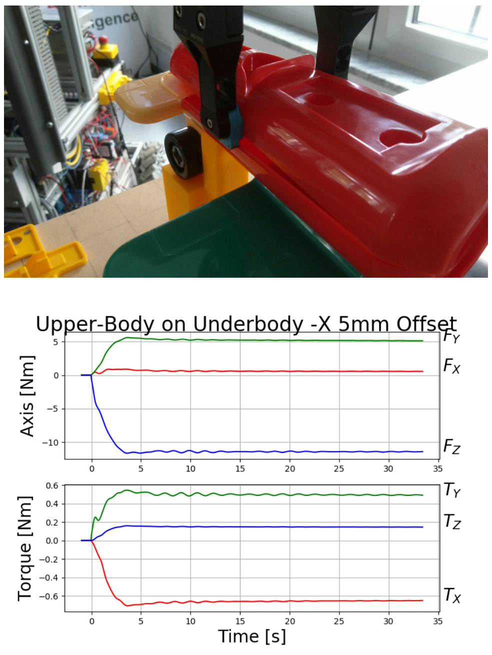


FIGURE A.10: Top: X (red), Y (green), Z (blue) orientation of the end effector while assembling the upper-body onto the underbody. Bottom: Heuristic's confusion matrix for this scenario.

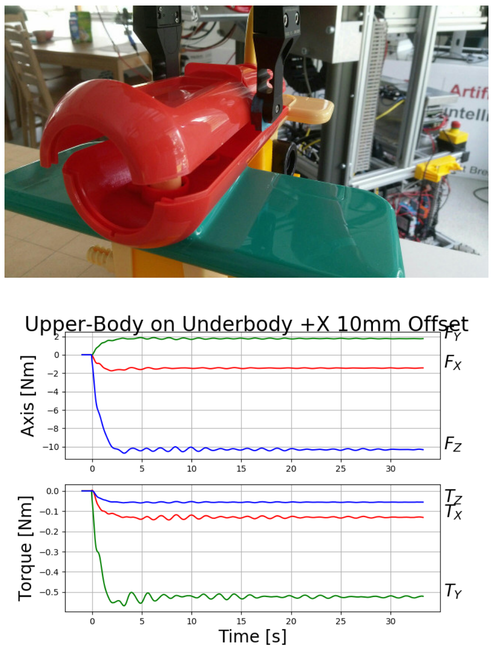


+X offset 5mm

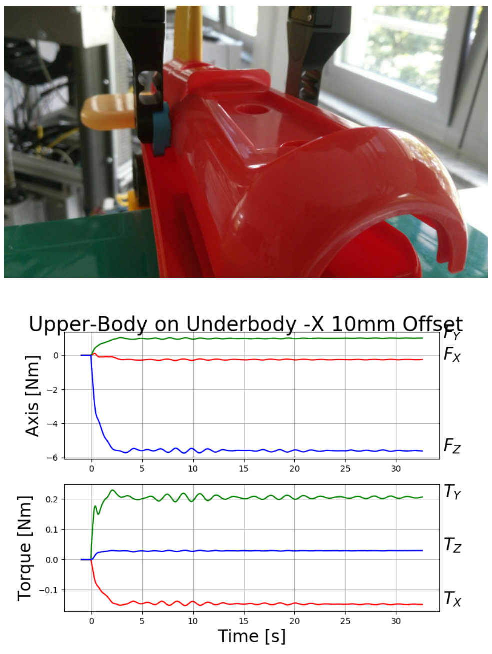


-X offset 5mm

FIGURE A.11: Upper-body on underbody forces with 5mm X offsets.



+X offset 10mm



-X offset 10mm

FIGURE A.12: Upper-body on underbody forces with 10mm X offsets.

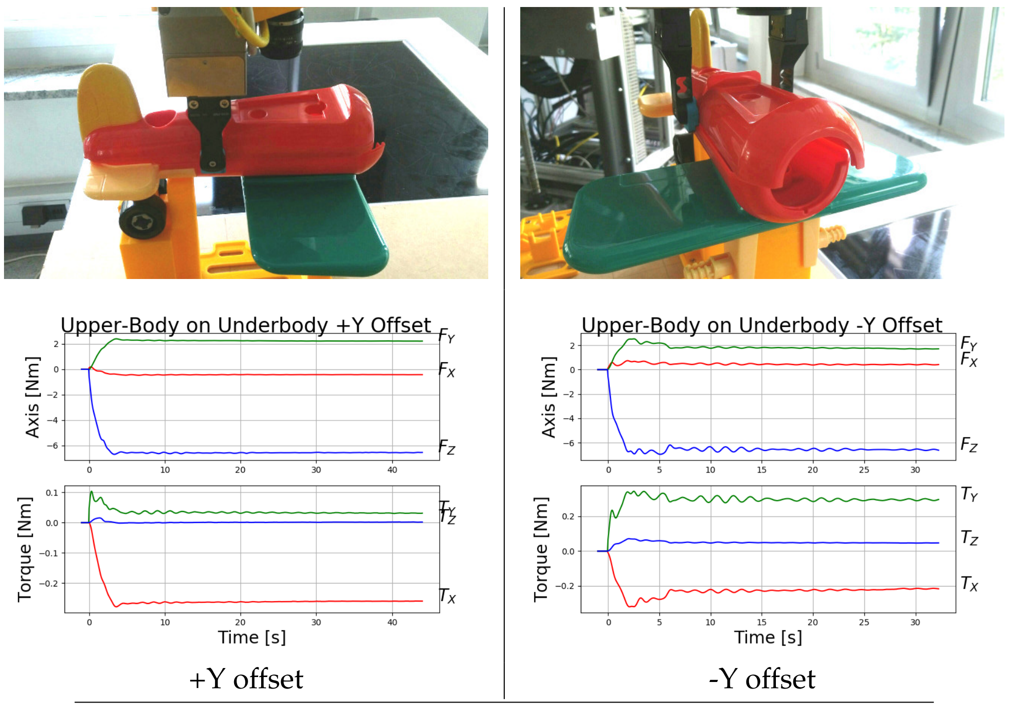


FIGURE A.13: Upper-body on underbody forces with Y offsets.

A.5 Bolt into rear hole

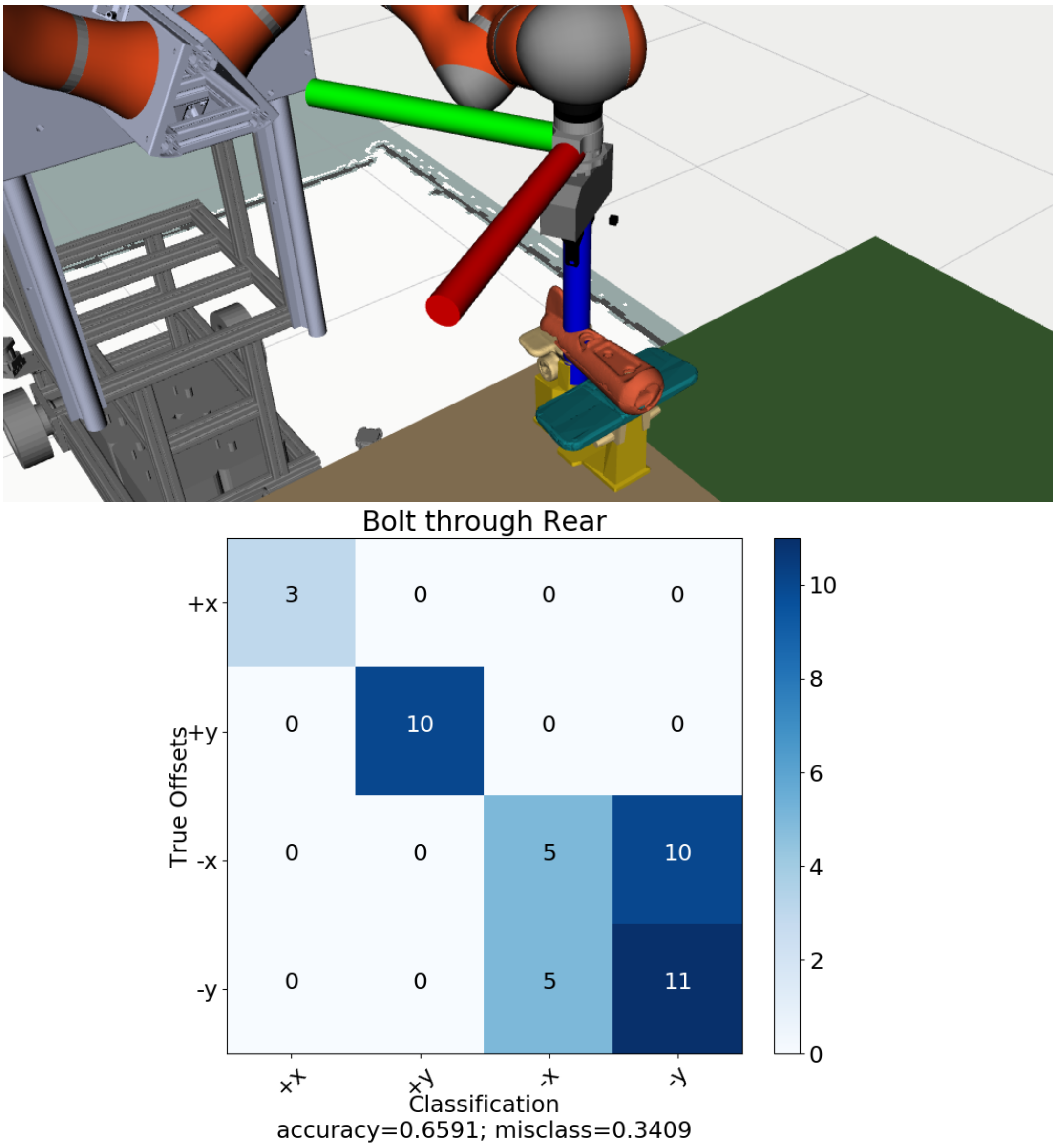


FIGURE A.14: Top: X (red), Y (green), Z (blue) orientation of the end effector while assembling the bolt into the rear hole. Bottom: Heuristic's confusion matrix for this scenario.

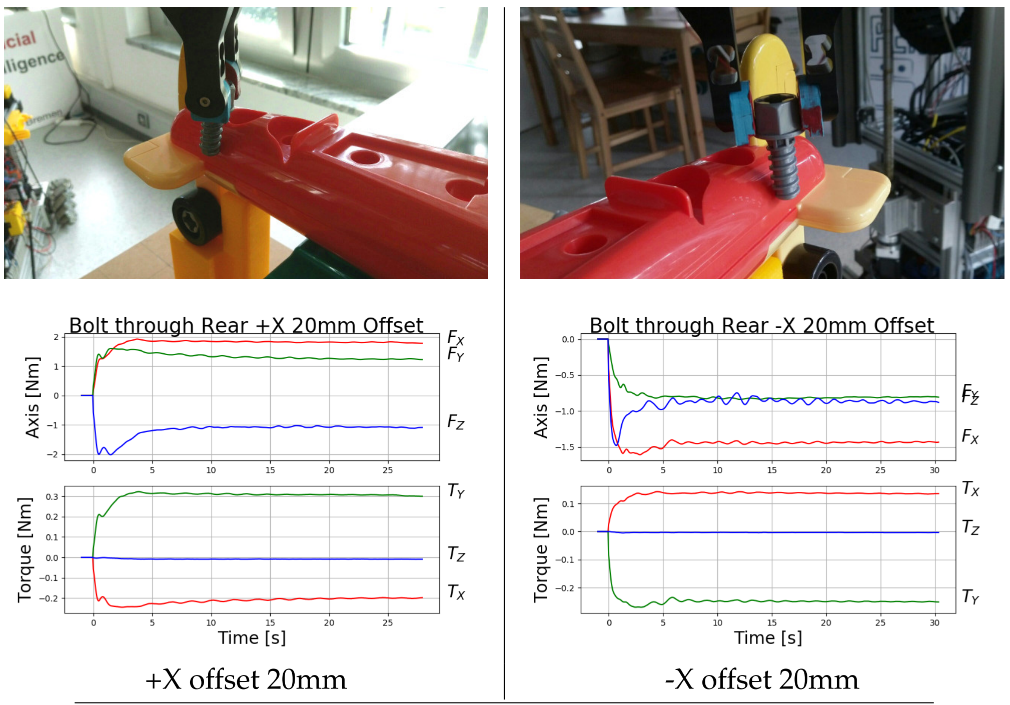
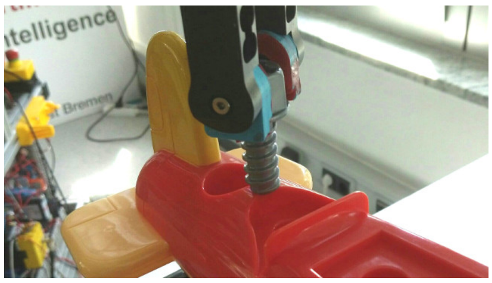
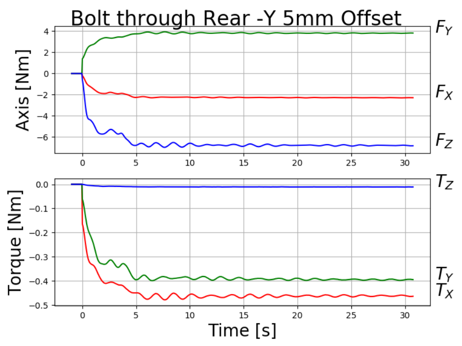
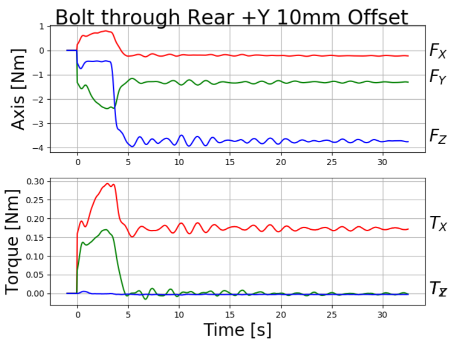


FIGURE A.15: Bolt in rear hole with X offsets.

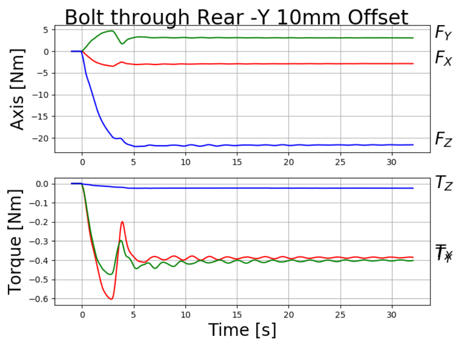




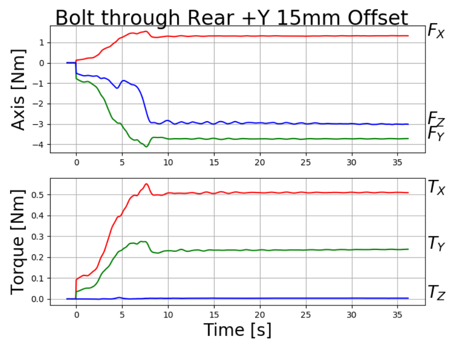
-Y offset 5mm



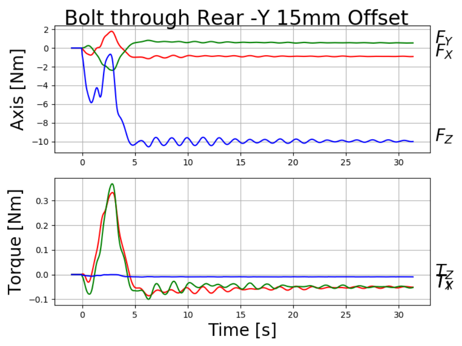
+Y offset 10mm



-Y offset 10mm



+Y offset 15mm



-Y offset 15mm

FIGURE A.16: Bolt in rear hole with Y offsets.

Bibliography

Bohren, J. et al. (2011). “Towards autonomous robotic butlers: Lessons learned with the PR2”. In: *2011 IEEE International Conference on Robotics and Automation*, pp. 5568–5575. DOI: 10.1109/ICRA.2011.5980058.

Bouchard, C. et al. (2015). “6D frictional contact for rigid bodies”. In: *Proceedings of the 41st Graphics Interface Conference, Halifax, NS, Canada, June 3-5, 2015*. Ed. by Hao (Richard) Zhang and Tony Tang. ACM, pp. 105–114. URL: http://dl.acm.org/citation.cfm?id=2788910.

Bruyninckx, Herman, Stefan Dutre, and Joris De Schutter (1995). “Peg-on-hole: a model based solution to peg and hole alignment”. In: *Proceedings of 1995 IEEE International Conference on Robotics and Automation*. Vol. 2. IEEE, pp. 1919–1924.

Burger, Robert et al. (2010). “The driver concept for the DLR lightweight robot III”. In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems, October 18-22, 2010, Taipei, Taiwan*. IEEE, pp. 5453–5459. DOI: 10.1109/IROS.2010.5650299. URL: https://doi.org/10.1109/IROS.2010.5650299.

Calli, B. et al. (2015). “The YCB object and Model set: Towards common benchmarks for manipulation research”. In: *2015 International Conference on Advanced Robotics (ICAR)*, pp. 510–517. DOI: 10.1109/ICAR.2015.7251504.

Chaves-Arbaiza, I., D. García-Vaglio, and F. Ruiz-Ugalde (July 2018). “Smart Placement of a Two-Arm Assembly for An Everyday Object Manipulation Humanoid Robot Based on Capability Maps”. In: *2018 IEEE International Work Conference on Bioinspired Intelligence (IWOBI)*, pp. 1–9. DOI: 10.1109/IWOBI.2018.8464192.

Coumans, Erwin (2015). “Bullet physics simulation”. In: *Special Interest Group on Computer Graphics and Interactive Techniques Conference, SIGGRAPH ’15, Los Angeles, CA, USA, August 9-13, 2015, Courses*. ACM, 7:1. DOI: 10.1145/2776880.2792704. URL: https://doi.org/10.1145/2776880.2792704.

Cousins, S. (2010). “ROS on the PR2 [ROS Topics]”. In: *IEEE Robotics Automation Magazine* 17.3, pp. 23–25. DOI: 10.1109/MRA.2010.938502.

García-Vaglio D. Ruiz-Ugalde, F. (Oct. 2020). *GPU based approach for fast generation of robot capability representations*.

Haidu, Andrei et al. (2018). “KNOWROB-SIM — Game Engine-enabled Knowledge Processing for Cognition-enabled Robot Control”. In: *International Conference on Intelligent Robots and Systems (IROS)*. IEEE. Madrid, Spain.

Hiemstra, P and A Nederveen (2007). “Monte carlo localization”. In: *Ad Hoc Networks* 6.5, pp. 718–733.

Hietanen, Antti et al. (2020). “AR-based interaction for human-robot collaborative manufacturing”. In: *Robotics and Computer-Integrated Manufacturing* 63, p. 101891.

Hirzinger, Gerd et al. (2002). “DLR’s Torque-Controlled Light Weight Robot III - Are We Reaching the Technological Limits Now?” In: *Proceedings of the 2002 IEEE International Conference on Robotics and Automation, ICRA 2002, May 11-15, 2002, Washington, DC, USA*. IEEE, pp. 1710–1716. DOI: 10.1109/ROBOT.2002.1014788. URL: https://doi.org/10.1109/ROBOT.2002.1014788.

Inoue, Tadanobu et al. (2017). “Deep reinforcement learning for high precision assembly tasks”. In: *2017 IEEE/RSJ International Conference on Intelligent Robots and*

Systems, IROS 2017, Vancouver, BC, Canada, September 24-28, 2017. IEEE, pp. 819–825. DOI: 10.1109/IROS.2017.8202244. URL: https://doi.org/10.1109/IROS.2017.8202244.

Jiang, Tao et al. (2020). "A measurement method for robot peg-in-hole pre-alignment based on combined two-level visual sensors". In: *IEEE Transactions on Instrumentation and Measurement*.

Kazhoyan, G. and M. Beetz (2019). "Executing Underspecified Actions in Real World Based on Online Projection". In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5156–5163. DOI: 10.1109/IROS40897.2019.8967867.

Kazhoyan, Gayane and Michael Beetz (2017). "Programming robotic agents with action descriptions". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2017, Vancouver, BC, Canada, September 24-28, 2017*. IEEE, pp. 103–108. ISBN: 978-1-5386-2682-5. DOI: 10.1109/IROS.2017.8202144. URL: https://doi.org/10.1109/IROS.2017.8202144.

Khansari-Zadeh, Seyed Mohammad, Ellen Klingbeil, and Oussama Khatib (2016). "Adaptive human-inspired compliant contact primitives to perform surface-surface contact under uncertainty". In: *Int. J. Robotics Res.* 35.13, pp. 1651–1675. DOI: 10.1177/0278364916648389. URL: https://doi.org/10.1177/0278364916648389.

Kumar, Deepesh et al. (2020). "Neuromorphic Approach to Tactile Edge Orientation Estimation using Spatiotemporal Similarity". In: *Neurocomputing*.

Kyung-Lyong Han et al. (2009). "Design and control of mobile robot with Mecanum wheel". In: *2009 ICCAS-SICE*, pp. 2932–2937.

Lee, YeonSun et al. (Nov. 2018). "Quantitative Comparison of Acupuncture Needle Force Generation According to Diameter". In: *Journal of Acupuncture Research* 35, pp. 238–243. DOI: 10.13045/jar.2018.00283.

Liu, Nailong et al. (2020). "Learning peg-in-hole assembly using Cartesian DMPs with feedback mechanism". In: *Assembly Automation*.

Luo, Shan et al. (2017). "Robotic Tactile Perception of Object Properties: A Review". In: *CoRR* abs/1711.03810. arXiv: 1711.03810. URL: http://arxiv.org/abs/1711.03810.

Maldonado, Alexis, Ulrich Klank, and Michael Beetz (2010). "Robotic grasping of unmodeled objects using time-of-flight range data and finger torque information". In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, pp. 2586–2591.

Mösenlechner, Lorenz (2016). "The Cognitive Robot Abstract Machine: A Framework for Cognitive Robotics". PhD thesis. Technical University Munich, Germany. URL: http://nbn-resolving.de/urn:nbn:de:bvb:91-diss-20160520-1239461-1-3.

Muxfeldt, Arne and Daniel Kubus (2016). "Hierarchical decomposition of industrial assembly tasks". In: *21st IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2016, Berlin, Germany, September 6-9, 2016*. IEEE, pp. 1–8. DOI: 10.1109/ETFA.2016.7733742. URL: https://doi.org/10.1109/ETFA.2016.7733742.

Mösenlechner, L. and M. Beetz (2013). "Fast temporal projection using accurate physics-based geometric reasoning". In: *2013 IEEE International Conference on Robotics and Automation*, pp. 1821–1827. DOI: 10.1109/ICRA.2013.6630817.

Newman, Wyatt S., Yonghong Zhao, and Yoh-Han Pao (2001). "Interpretation of Force and Moment Signals for Compliant Peg-in-Hole Assembly". In: *Proceedings of the 2001 IEEE International Conference on Robotics and Automation, ICRA*

2001, May 21-26, 2001, Seoul, Korea. IEEE, pp. 571–576. DOI: 10.1109/ROBOT.2001.932611. URL: https://doi.org/10.1109/ROBOT.2001.932611.

Park, Hyeonjun et al. (2020). "Compliant Peg-in-Hole Assembly Using Partial Spiral Force Trajectory With Tilted Peg Posture". In: *IEEE Robotics and Automation Letters* 5.3, pp. 4447–4454.

Quigley, Morgan et al. (2009). "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software*.

Roth, Scott D. (1982). "Ray casting for modeling solids". In: *Comput. Graph. Image Process.* 18.2, pp. 109–144. DOI: 10.1016/0146-664X(82)90169-1. URL: https://doi.org/10.1016/0146-664X(82)90169-1.

Ruiz-Ugalde, F., G. Cheng, and M. Beetz (2011). "Fast adaptation for effect-aware pushing". In: *2011 11th IEEE-RAS International Conference on Humanoid Robots*, pp. 614–621. DOI: 10.1109/Humanoids.2011.6100863.

Schoettler, Gerrit et al. (2019). *Deep Reinforcement Learning for Industrial Insertion Tasks with Visual Inputs and Natural Rewards*. arXiv: 1906.05841 [cs.RO].

Schoettler, Gerrit et al. (2020). "Meta-Reinforcement Learning for Robotic Industrial Insertion Tasks". In: *arXiv preprint arXiv:2004.14404*.

Sharma, K., V. Shirwalkar, and P. K. Pal (2013). "Intelligent and environment-independent Peg-In-Hole search strategies". In: *2013 International Conference on Control, Automation, Robotics and Embedded Systems (CARE)*, pp. 1–6. DOI: 10.1109/CARE.2013.6733716.

Stelter, Simon, Georg Bartels, and Michael Beetz (2018). "Multidimensional Time-Series Shapelets Reliably Detect and Classify Contact Events in Force Measurements of Wiping Actions". In: *IEEE Robotics Autom. Lett.* 3.1, pp. 320–327. DOI: 10.1109/LRA.2017.2716423. URL: https://doi.org/10.1109/LRA.2017.2716423.

Sun, Yi et al. (2020). "Learn How to Assist Humans Through Human Teaching and Robot Learning in Human-Robot Collaborative Assembly". In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems*.

Taddeucci, Davide et al. (1997). "An approach to integrated tactile perception". In: *Proceedings of the 1997 IEEE International Conference on Robotics and Automation, Albuquerque, New Mexico, USA, April 20-25, 1997*. IEEE, pp. 3100–3105. DOI: 10.1109/ROBOT.1997.606759. URL: https://doi.org/10.1109/ROBOT.1997.606759.

Takahashi, Tomoichi and Hiroyuki Ogata (1992). "Robotic assembly operation based on task-level teaching in virtual reality". In: *Proceedings 1992 IEEE International Conference on Robotics and Automation*. IEEE Computer Society, pp. 1083–1084.

Tsujimura, T. and T. Yabuta (1989). "Object detection by tactile sensing method employing force/torque information". In: *IEEE Transactions on Robotics and Automation* 5.4, pp. 444–450. DOI: 10.1109/70.88059.

Wong, Percy Charles (1975). "Peg-hole assembly; an investigation into tactile methods". PhD thesis. University of Canterbury. Mechanical Engineering.

Yamamoto, Takashi et al. (2018). "Human support robot (HSR)". In: *ACM SIGGRAPH 2018 emerging technologies*, pp. 1–2.

Yang, L. et al. (2020). "Rigid-Soft Interactive Learning for Robust Grasping". In: *IEEE Robotics and Automation Letters* 5.2, pp. 1720–1727. DOI: 10.1109/LRA.2020.2969932.