Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

MASTERTHESIS

Adaptive Pouring of Liquids Based on Human Motions Using a Robotic Arm

submitted by

Jeremias Kornelius Hartz

Fakulty:	MIN-Faculty
Major:	Information Systems
Matriculation No.:	6424817
Primary Supervisor:	Prof. Dr. Jianwei Zhang
Secondary Supervisor:	Dr. Norman Hendrich
Advisor:	Michael Görner
Date of Submission:	18.07.2018

Abstract

Pouring of fluids is a complicated task in robotics that requires a lot of information about the environment. It is therefore often implemented in a static way in order not to deal with its whole complexity.

In this master thesis a concept and the partial development of a multimodal adaptive pouring behavior is presented using a 3-finger gripper and a force torque sensor attached to a robotic arm, and a camera. Furthermore, human pouring motions are recorded and a tool implemented to analyze their characteristics and to play them back on the robot.

The result of this thesis is a realizable concept for adaptive liquid pouring with practical solutions and an advanced prototype. The prototype can be operated through a web-based user interface for analyzing and executing motions.

Zusammenfassung

Das Eingießen von Flüssigkeiten ist eine komplizierte Tätigkeit in der Robotik, die viele Informationen über die Umwelt benötigt. Häufig werden bei dessen Implementierung statische Verhalten zur Vereinfachung eingebaut.

In dieser Masterarbeit wird ein Konzept und eine Teilentwicklung eines multimodalen, adaptiven Eingießverhaltens präsentiert und auf einen Roboterarm mit einem 3-Finger Greifer und einem Kraft/Drehmoment-Sensor, und einer Kamera ausgeführt. Außerdem werden menschliche Eingießbewegungen aufgenommen und ein Tool zur Analyse ihrer Charakteristiken und zu ihrem Abspielen auf dem Roboter entwickelt.

Das Ergebnis dieser Arbeit ist ein realisierbares Konzept zum adaptiven Eingießen von Flüssigkeiten mit praktischen Lösungen und ein fortgeschrittener über eine webbasierte Benutzeroberfläche steuerbarer Prototyp zum Analysieren und Ausführen von Bewegungen.

List of Contents

List of Figures

List of Tables

Glossary

3D three-dimensional. 1, 5, 7, 8, 11, 13, 15, 19, 21

API application programming interface. 10

CCP `computeCartesianPath` is a method generating a robot trajectory from given waypoints. 12, 13, 31–34, 49, 50, 53, 58, 59

CSV comma-separated values. 25, 28, 33, 59

DOF degrees of freedom. 7

End effector is the device at the end of a robotic arm, used to interact with the environment. 7, 12, 17, 34

GUI graphical user interface. 11, 18

HTML hypertext markup language. 36, 40

IK inverse kinematics. 7, 30–32, 52

MoveIt! is a robot planning framework for ROS. 11–13, 17, 18, 29, 30, 32, 33, 57

ROS Robot Operating System. 5, 7, 10, 11, 22, 35, 36

Rviz ROS visualization. 11, 25, 28, 29

URDF unified robot description format. 5, 11, 13, 18, 19, 26, 32, 59

WUI web user interface. 25, 34, 36, 41, 61

Xacro XML macros. 11

XML extensible markup language. 10, 11

1 Introduction

Robotic arms have revolutionized manufacturing expanding rapidly in production lines throughout the years [1]. Using robotic arms to pour liquids is a task that is challenged by the complex physical properties of liquids combined with real time computation requirements.

Robotic pouring is used mostly in manufacturing applications (e.g., pouring molten metal [2], pipetting [3]) using repetitive motions that have to be reprogrammed when small changes in the environment occur. Such environments are usually specifically designed to simplify the pouring process for robots. Most robotic operations with liquids are done with liquid dispensers sometimes being attached to robotic arms, but without grippers that tilt pouring containers.

Outside of manufacturing, robotic arms are not used as frequently due to the price and the high running costs for experts maintaining them. Pouring liquids with grippers holding a bottle is mainly used for entertaining purposes today because of its imprecision and slowness.

This thesis is built on a preceding university group project that developed a bartender demo where a robot recognizes bottles, grabs them, and pours liquids into a glass. The pouring motion in the demo is always the same and stops midst motion for a time depending on the amount being poured. It is neither efficient nor precise which is why the focus lies on the analysis of pouring motions and their adaptation. This is done by recording human pouring motions.

1.1 Outline

This first chapter describes the motivation for research in robotic pouring due to challenging tasks like cooking in section 1.2. Section 1.3 lists the research questions and the goals of the thesis. Section 1.4 closes the chapter introducing the robot used for pouring and its parts.

Chapter 2 covers basic mathematical concepts needed for understanding the control of a robotic arm in three-dimensional (3D) space and the software used for the implementation in chapter 6.

Chapter 3 introduces multiple ways of pouring with robots and what is done by other researchers.

Chapter 4 explains the adaptive pouring concept of this thesis in theory and with practical solution proposals for different components.

Chapter 5 describes the laboratory experiment conducted for recording human pouring motions.

Chapter 6 goes through the implementation of a prototype for analysis and pouring execution.

Chapter 7 presents and discusses the results from chapter 5 and chapter 6.

Chapter 8 concludes the results, difficulties during the course of the thesis and gives suggestions for future research.

1.2 Motivation

Robotic arms can become common in restaurants and households as more tools for easier usage are developed, their range of executable tasks grows, and their price decreases. Pouring robots have to become more adaptive to work in these environments and collaborate with humans. Examples for applications of the future are cooking, cleaning or even feeding. Research is needed to advance in this topic that has the potential to bring robotics another step forward.

1.3 Questions and Goals

The main question behind this thesis is how the task of consistently pouring liquids into a glass in different environments with a robotic arm can be solved. The first goal of the thesis is to present a concept for implementing a pouring behavior that deals with different environments and define the changes in environments that influence it.

The follow up question is how human pouring motions look like and if robots can imitate them successfully. The second goal of the thesis is therefore creating a plan for conducting an experiment to record human pouring motions and to implement a tool for analyzing and replaying them on a robot.

1.4 Robot

The robot used for the pouring task consists of:

1. UR5 collaborative robot arm [4]
   This robotic arm has a 5 kg payload and a reach of 85 cm.
2. Force Torque Sensor FT 150 [5]
   Range of ±150 Newton in steps of 0.2 Newton
3. Wireless Bluetooth system [6]
4. 3-Finger Adaptive Robot Gripper [7]
5. Camera (with depth sensor)[8]

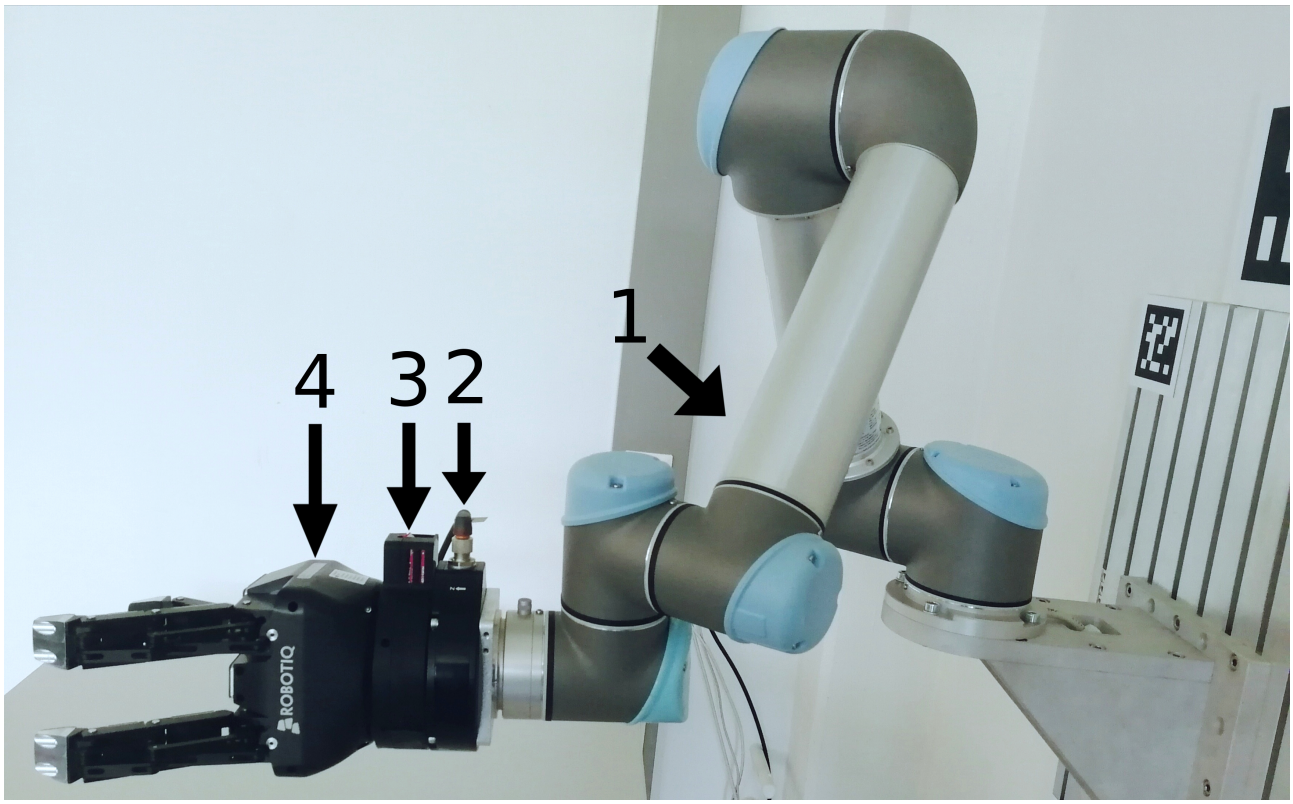


Figure 1.1: The parts of the robot: UR5 (1), FT 150 (2), wireless Bluetooth system (3), gripper (4)

The force torque sensor, the wireless Bluetooth system and the gripper are attached to the robot arm in that order (fig. 1.1). The wireless Bluetooth system is made to get rid of the long cables attached to the force torque sensor and the gripper. The cables often caused problems during movements, e.g., knocking over the glass or other bottles. The camera is mounted on a tripod (fig. 1.2), viewing the bottles and AprilTags (section 2.7) on the wall seen in fig. 1.3.

Figure 1.2: Camera recognizing and locating bottles

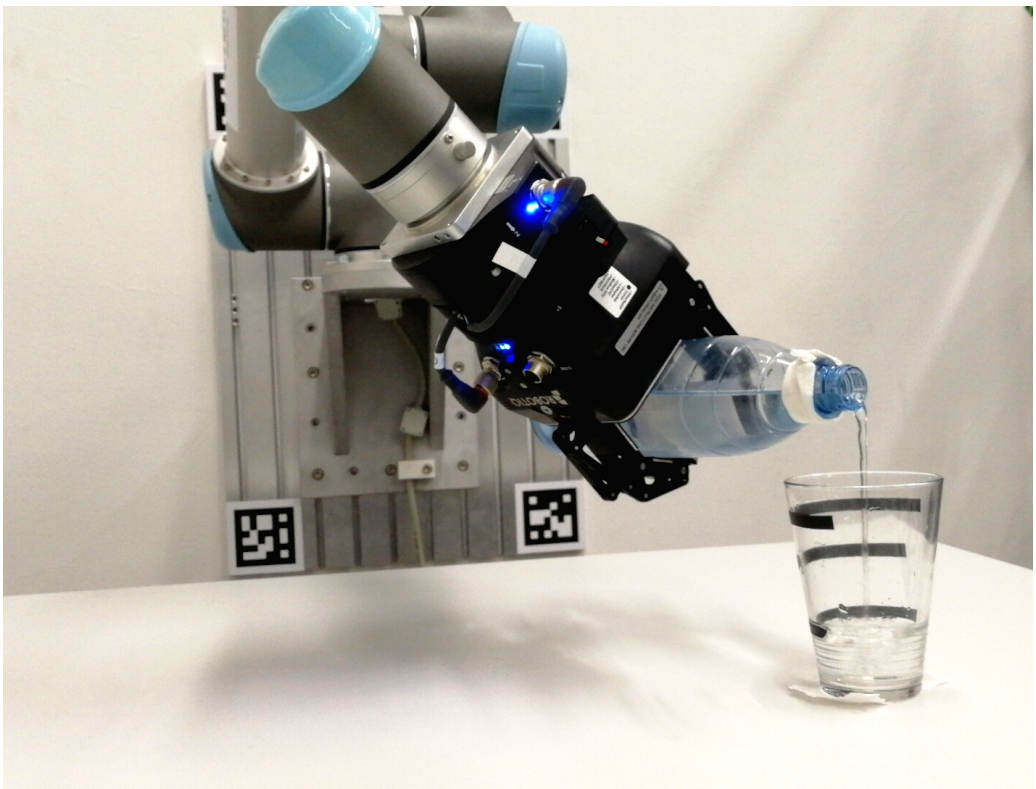


Figure 1.3: Pouring test

2 Fundamentals

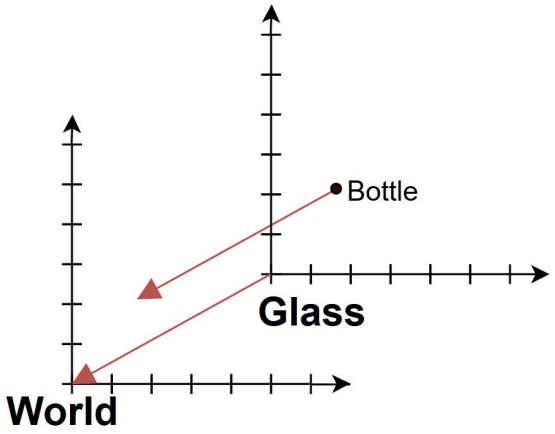
This chapter covers specific aspects of theory, robotics and software needed to understand the other chapters of this thesis. The sections in this chapter build on one another.

In section 2.1 basic theoretic concepts for setting target coordinates in 3D coordinate systems are explained. Section 2.2 describes the theory behind moving a robotic arm. Section 2.3 explains the most important parts of the Robot Operating System (ROS), the software used for implementing the prototype, while going into detail at parts that are needed to understand the implementation in chapter 6. The following section 2.4 explains unified robot description format (URDF) files used in ROS and how they are structured. Section 2.5 and section 2.6 describe the two main frameworks used for moving the robotic arm. Lastly section 2.7 explains AprilTags that are used for locating objects.

2.1 Transformations

When programming a moving robot arm, targets are often defined as points in a coordinate system. In the robot environment there are different coordinate systems, also called frames (e.g., world, table, robot,...) and it is often necessary to map given coordinate points to other coordinate systems. This can be done using transformations. A transformation can consist of a translation, rotation or scaling. Scaling is not needed for this thesis as the objects are not growing or shrinking. A 3D transformation is often represented by a 4D matrix, a so called homogeneous transformation matrix, because it enables multiple transformations with one operation.

The examples in fig. 2.1 and fig. 2.2 demonstrate how transformations are applied when mapping a recorded bottle point in the world frame to the glass frame, given the transformation from the world to the glass frame. The first example shows a simple translation, the second one a translation and a rotation of the bottle point.



$$Glass_{world} = (x_G, y_G, z_G) \quad Bottle_{world} = (x_B, y_B, z_B)$$

$$Bottle_{Glass} = \begin{bmatrix} 1 & 0 & 0 & -x_G \\ 0 & 1 & 0 & -y_G \\ 0 & 0 & 1 & -z_G \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_B \\ y_B \\ z_B \\ 1 \end{bmatrix} = \begin{bmatrix} x_B - x_G \\ y_B - y_G \\ z_B - z_G \\ 1 \end{bmatrix}$$

Figure 2.1: Translation example: The bottle point is transformed using the translation from the glass to the world frame resulting in the coordinates of the bottle in the glass frame.

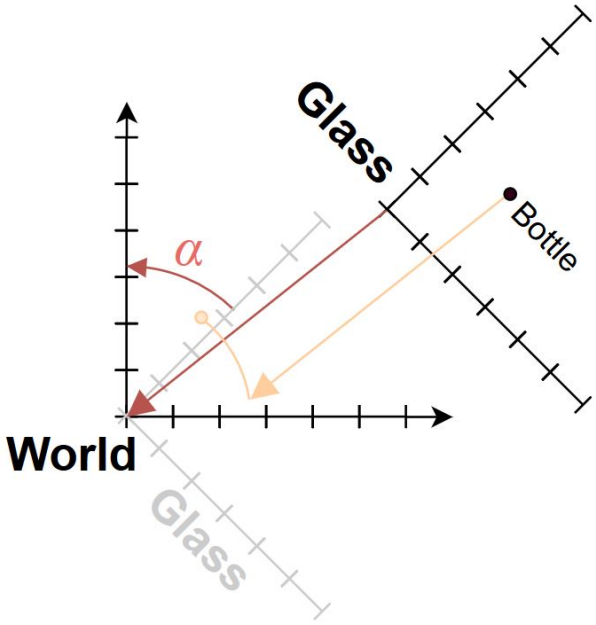


Figure 2.2: Translation and rotation example visualization: First the translation and then the rotation is applied to bring the glass frame to the world frame (equation in fig. 2.4). A rotation and then a translation can be applied too to reach the same state but only with different values. Changing the order of transformations without adjusting the values leads to different results.

There are three rotation matrices corresponding to the three axes where a positive value of angle α leads to a counterclockwise rotation (fig. 2.3).

$$X \rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad Y \rightarrow \begin{bmatrix} \cos(\alpha) & 0 & \sin(\alpha) & 1 \\ 0 & 1 & 0 & 1 \\ -\sin(\alpha) & 0 & \cos(\alpha) & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad Z \rightarrow \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 2.3: Rotation matrices for each axis

A translation and a rotation can be written in one matrix if the translation happens first (fig. 2.4). It is important to keep the correct order of transformations because matrix multiplication is not commutative. Transformations are applied from right to left when using a column vector for the point being transformed.

$$\begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & -x_G \\ \sin(\alpha) & \cos(\alpha) & 0 & -y_G \\ 0 & 0 & 1 & -z_G \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_B \\ y_B \\ z_B \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\alpha) \times x_B - \sin(\alpha) \times y_B - x_G \\ \sin(\alpha) \times x_B + \cos(\alpha) \times y_B - y_G \\ z_B - z_G \\ 1 \end{bmatrix}$$

Figure 2.4: A translation and rotation in a single transformation matrix (left) must be performed in that order.

When applying a transformation including a rotation to a `Pose` (listing 2.3) and its orientation the rotation should be applied to the orientation vector too.

2.2 Inverse Kinematics

Algorithms computing inverse kinematics (IK) are applied when finding the values of a robot's joints to reach a specified target pose with its end effector. In most cases there is more than one solution which makes it more complex than forward kinematics where the target pose is unknown and computed by the given joint values, always resulting in exactly one solution. There are multiple ways of solving an IK problem and the complexity grows with increasing degrees of freedom (DOF). DOF are the number of independent displacements of the robot. Generally IK algorithms are categorized into two categories:

1. **Analytical solvers**
   Analytical solvers are fast and give the same solutions repeatably. They have to be adjusted anytime a new DOF is added or changed and become increasingly complex with more DOF. For most robotic arms such as the one used in this thesis an analytical solver is the most reliable one.
2. **Numerical solvers**
   Analytical solvers do not always suffice, e.g., when moving animations with over 100 DOF, which is when numerical solvers can be utilized well. Numerical algorithms do not always find a solution and the solution may differ each time. They use approximation which allows them to find a solution that differs from the goal but this can be especially useful in cases where the solution specified is not reachable or does not need to be extremely precise.

The two categories are not always completely separable as approaches exist where both of them are used [9].

2.3 Robot Operating System (ROS)

ROS is an open source software framework for programming robots in mostly C++ and Python [10]. It separates the robot hardware from the software part which enables reusing the same code on different robots. ROS provides a modular architecture for building interchangeable parts of software. In the following subsections its components relevant in this thesis are introduced.

2.3.1 Package

Packages are the modules of ROS, each package represents a useful module that can easily be reused. It needs to have a distinct name to avoid naming collisions because it must be referenced every time it is used in other packages. Packages are the smallest component that can be built in ROS and need to follow a standard structure [11]. In the following an overview of some important packages is given and how they are used in this thesis.

- `tams_pour` is the package created specifically for this thesis to extract, analyze and execute recorded human pouring motions.
- `tams_ur5_bartender` was made for a preceding university group project and is used for bottle recognition, grasping bottles, and executing a simple pouring motion that is replaced by the `tams_pour` implementation in this thesis.
- `moveit` is used for moving the arm, and described in section 2.5.
- `geometry_msgs` is used for standard geometrical messages (section 2.3.2).
- `tf` (new version: `tf2`) is used for transforming geometrical message in 3D space.

2.3.2 Message

A message is a structure containing information, much like an object in object-oriented programming languages. To create, send or receive a message, a file defining the message type must be created and included in the code file. If a property is not of a built-in type (list of available types in [12], part 2.1.1) its package has to be declared as well. The format of each property is `(<package>/)<type> <name>`. The most important message types that are used in this thesis are introduced below.

Listing 2.1: geometry_msgs/Point.msg

```
float64 x
float64 y
float64 z
```

The `Point` message describes a point in a 3D coordinate system. The unit of x, y and z is meter (m).

Listing 2.2: geometry_msgs/Quaternion.msg

```
float64 x
float64 y
float64 z
float64 w
```

The `Quaternion` message describes a quaternion which represents an orientation in a 3D coordinate system. Quaternions are a standard in robotics and more efficient in many aspects (e.g., no gimbal lock) than other orientation conventions [13, 14], but not as intuitive for most humans as Euler angles, an orientation convention consisting of three components: ϕ (roll), θ (pitch) and ψ (yaw). These components represent the rotation around the x, y and z axis in the unit radian (rad). The conversions between degrees and radians are shown below.

$$degrees = radians \times \frac{180}{\pi} \qquad radians = degrees \times \frac{\pi}{180}$$

Following equations from [15] show the conversions between a quaternion, q, and Euler angles, (ϕ, θ, ψ) used in this thesis; beware that other rotation sequences exists which will not work correctly with these equations and that singularities must be handled separately.

$$\begin{bmatrix}\phi\\ \theta\\ \psi\end{bmatrix} = \begin{bmatrix}\arctan(\frac{2(wx+yz)}{1-2(x^2+y^2)})\\ \arcsin(2(wy+zx))\\ \arctan(\frac{2(wz+xy)}{1-2(y^2+z^2)})\end{bmatrix} \qquad q\begin{bmatrix}x\\ y\\ z\\ w\end{bmatrix} = \begin{bmatrix}\sin(\phi/2)\cos(\theta/2)\cos(\psi/2)+\cos(\phi/2)\sin(\theta/2)\sin(\psi/2)\\ \cos(\phi/2)\sin(\theta/2)\cos(\psi/2)-\sin(\phi/2)\cos(\theta/2)\sin(\psi/2)\\ \cos(\phi/2)\cos(\theta/2)\sin(\psi/2)+\sin(\phi/2)\sin(\theta/2)\cos(\psi/2)\\ \cos(\phi/2)\cos(\theta/2)\cos(\psi/2)-\sin(\phi/2)\sin(\theta/2)\sin(\psi/2)\end{bmatrix}$$

For converting x, y, z rotation angles to quaternions, the method `createQuaternionMsgFromRollPitchYaw(r, p, y)` from the package `tf` can be used [16]. Converting them back is not as simple (mentioned in the last chapter in section 8.2.1.1).

Listing 2.3: geometry_msgs/Pose.msg

```
geometry_msgs/Point position
geometry_msgs/Quaternion orientation
```

The `Pose` message combines a `Point` and a `Quaternion`.

Listing 2.4: std_msgs/Header.msg

```
uint32 seq
time stamp
string frame_id
```

The `Header` message is used for identification. It holds an id (`seq`), a timestamp (`stamp`) and the name of the frame (`frame_id`) it lies in. The latter is useful for transforming poses between different frames.

Listing 2.5: geometry_msgs/PoseStamped.msg

```
std_msgs/Header header
geometry_msgs/Pose pose
```

The `PoseStamped` message combines a `Pose` and a `Header`.

Listing 2.6: tams_pour/PoseStampedArray.msg

```
std_msgs/Header header
geometry_msgs/PoseStamped[] data
```

The `PoseStampedArray` message combines a `Header` and an array of `PoseStamped`. It is created for sending entire motions.

Listing 2.7: tams_pour/Trajectory.msg

```
tams_pour/PoseStampedArray stampedPoses
bool bottleSpout
bool high
bool person
bool slow
bool valid
int32 initialAmount
int32 pouredAmount
```

The `Trajectory` message combines a `PoseStampedArray` and multiple properties for more efficient analysis, sorting, comparison and filtering.

2.3.3 Topic

A topic is used for information streaming. Only one predefined type of messages can be streamed through a topic. Nodes can publish their messages to the same topic or subscribe to it in order to receive all messages being published to it. Topics are publicly available to all nodes.

2.3.4 Node

A node is a process that computes something. A robot is operated by multiple nodes who communicate with each other. All nodes must register at a master to find each other.

2.3.5 Master

A master enables communication between nodes. A node that wants to publish its computed data, registers its topics at the master. Other nodes can ask the master if there is a certain

registered topic. Once the master receives at least one request to listen to a registered topic it notifies the publishing node and the data transfer to all listening nodes can begin.

2.3.6 Launch

A robot application usually consists of many nodes. The tool Roslaunch is used for starting multiple nodes at the same time, also remote nodes on different servers can be started through it. It is uses extensible markup language (XML) files with a `.launch` extension for execution. These `launch` files can also include other `launch` files.

2.3.7 Service

Nodes can provide services which have a unique name and consist of a request and response message. Other nodes can then call the service by providing its name and the request message and receive a response when the service finished its task. For tasks that take a long time to complete and where the requesting node needs feedback about the current state of the task Actions are the preferred over services.

2.3.8 Action

Actions are similar to services but can additionally provide feedback and be canceled during the completion of a task. On the downside it takes longer to implement them. An action runs on an action server and is called by an action client. These two can be controlled from nodes by using their corresponding application programming interface (API).

2.3.9 Bridge

Rosbridge enables language-independent communication between an external program and the ROS environment by providing a JavaScript Object Notation (JSON) API [17]. Using the JavaScript Roslibjs library, users can communicate with ROS through a browser [18] by publishing and subscribing to topics, and calling actions and services.

2.3.10 Bag

For recording data, using the Rosbag package, a bag can be created by recording all messages published to specified topics. When it is replayed it publishes the recorded messages to their original topics. When receiving a bag from a different ROS project, it is advised to get a hold of any non-standard message types that were recorded in order to make it easier to use the data in nodes.

2.3.11 Rviz

ROS visualization (Rviz) is a 3D visualizing tool for displaying sensor data and state information from ROS. It can display robot description files (explained in section 2.4) and data from topics, e.g., camera image and depth point clouds. There are multiple plugins that enhance its functionality like remote controlling robots through it. It is used in this thesis to display and simulate the robot, its environment and the pouring motions.

2.4 Unified Robot Description Format (URDF)

The URDF defines the standard XML structure for describing a robot that ROS can interpret. A robot consists of links, which represent its parts, and joints, which represent the connections between the links (fig. 2.5). There are several types of joints, fig. 2.6 shows the most basic ones. The robotic arm used in this thesis only consists of revolute joints. In a URDF file the robot's joints and links are described in a hierarchical order. Other XML tags for visual components, collision objects and more configurations can also be defined. For example: using the `<visual>` tag the robot model can be graphically displayed in Rviz. As URDF files can grow large with increasingly complex robot and environment models they are often extended to XML macros (Xacro) files. Xacro can evaluate macros and write the results into the resulting URDF file. Some of its uses with examples are listed below.

Listing 2.8: Constants

```
<xacro:property name="const_name" value="1"/>
<!--Usage - "1" is inserted below-->
${const_name}
```

Listing 2.9: Calculations

```
${1+(height/2)}
```

Listing 2.10: Includes

```
<xacro:include filename="path_to_file" />
```

Including other files enables URDF files to be created as reusable modules, e.g., for loading different robot models into the same environment.

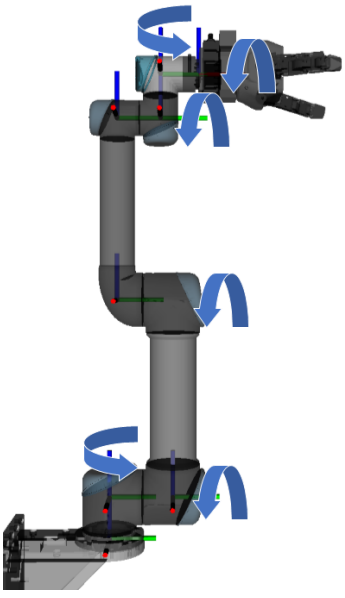


Figure 2.5: The robotic arm used in this thesis has eight links (coordinate frames indicate their origin), and six rotational joints (rotation arrows) not including the links and joints of the attached gripper.

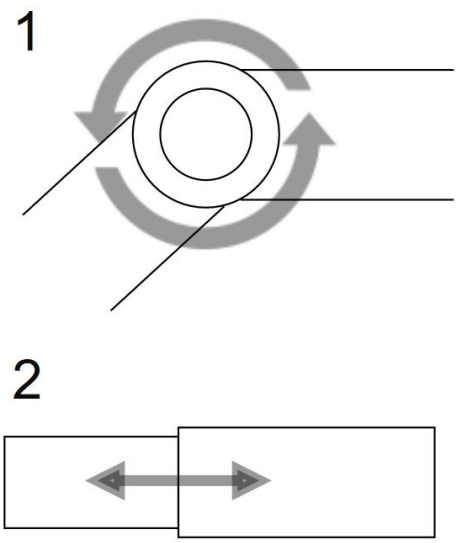


Figure 2.6: The two most basic joint types are revolute joints (1) and prismatic joints (2)

2.5 MoveIt!

MoveIt! is a software that runs on top of ROS and provides a framework for controlling robots to carry out manipulation tasks. It includes implementations for motion planning, manipulation, 3D perception, kinematics, control, and navigation, and utilizes some common ROS tools like Rviz and URDF. Its node `move_group` integrates all of the components and can be accessed through different interfaces. In this thesis the graphical user interface (GUI) `MotionPlanning`, a plugin for Rviz, is used for simulating the robot, its movements and for testing. The C++ interface `move_group_interface` is used for planning the robot's movements, retrieving information about its state, and moving it. Some the most used methods in the implementation of this thesis are explained in the following section 2.5.1, section 2.5.2 and section 2.5.3.

2.5.1 Set Start and Target

Two different types of targets can be set in MoveIt!.

- **Cartesian**
  The first type describes a `PoseStamped` for the end effector.
- **Joint**
  The second type describes the state of all joints of the robot.

The start state of a robot can only be set to a joint state, not a `PoseStamped`.

2.5.2 Move

The method `move` plans the robot's motion to reach a target from the start state (which can be changed to differ from the current state) and executes it. These two steps can also be split by using the methods `plan` and `execute` separately which is useful for visualizing the robot's motion before actually moving or planning ahead. The plan takes obstacles into account.

2.5.3 Compute Cartesian Path (CCP)

The last presented method, `computeCartesianPath` (CCP), receives a Cartesian path (a sequence of `Poses`) and computes the possible joint values, accelerations and velocities for every `Pose`. The result is referred to as a joint trajectory. All poses are taken into account for computing the speed values to achieve a smooth motion while moving as fast as possible. The method is used when the `Poses` in between the start and goal state are important (e.g., in a pouring motion). One of the limitations that the other planning methods in section 2.5.2 also have is that the time in which the targets have to be reached can not be set. The planned trajectory is based on the maximal speed values that are set in the robot's joint-configuration file which is loaded into the URDF. The speed limits can be changed during runtime using the methods `setMax[Velocity/Acceleration]ScalingFactor` but they can not be changed within a trajectory (unless doing so manually). A human pouring motion has changing speeds over time which is why MoveIt! alone is not enough and the Reflexxes library (section 2.6) is used in order to adjust the velocities and accelerations over time.

2.6 Reflexxes

The Reflexxes Motion Libraries [19] specialize on instantaneously generating smooth trajectories based on joint states and their limits. The focus lies on quick reactions to unforeseen events during manipulation tasks. Therefore the algorithm focuses on efficient trajectory computation. In this thesis its position-based algorithm is used in order to recalculate the speeds of a joint trajectory (returned by CCP) to match the original time-line of the corresponding recorded motion without jitter in the resulting trajectory.

Through the central method `ReflexxesAPI::RMLPosition` a smooth motion is always achieved but this can come with the cost of not reaching the targets at the desired time. The method's input parameter consists of three main parts.

- **Current State**
  The current state of the robot consists of its joint positions, velocities and accelerations.
- **Target State**
  The target state consists of the joint positions and velocities.
- **Limits**
  The limits describe the maximal velocities and accelerations of each joint, setting maximal jerk values is also possible, but not used in this work.

The method then computes joint positions, velocities and accelerations for the next state towards the target that can be reached in a previously set time.

2.7 AprilTag

AprilTags (fig. 2.7) are two-dimensional bar codes that encode between 4 and 12 bits of data, allowing them to be detected robustly and from long ranges. They are designed for high localization accuracy; their 3D position with respect to a camera can be precisely computed. AprilTags and their recognition algorithms are actively being further improved and a new version more stable has been introduced in 2016, AprilTag 2 [20].

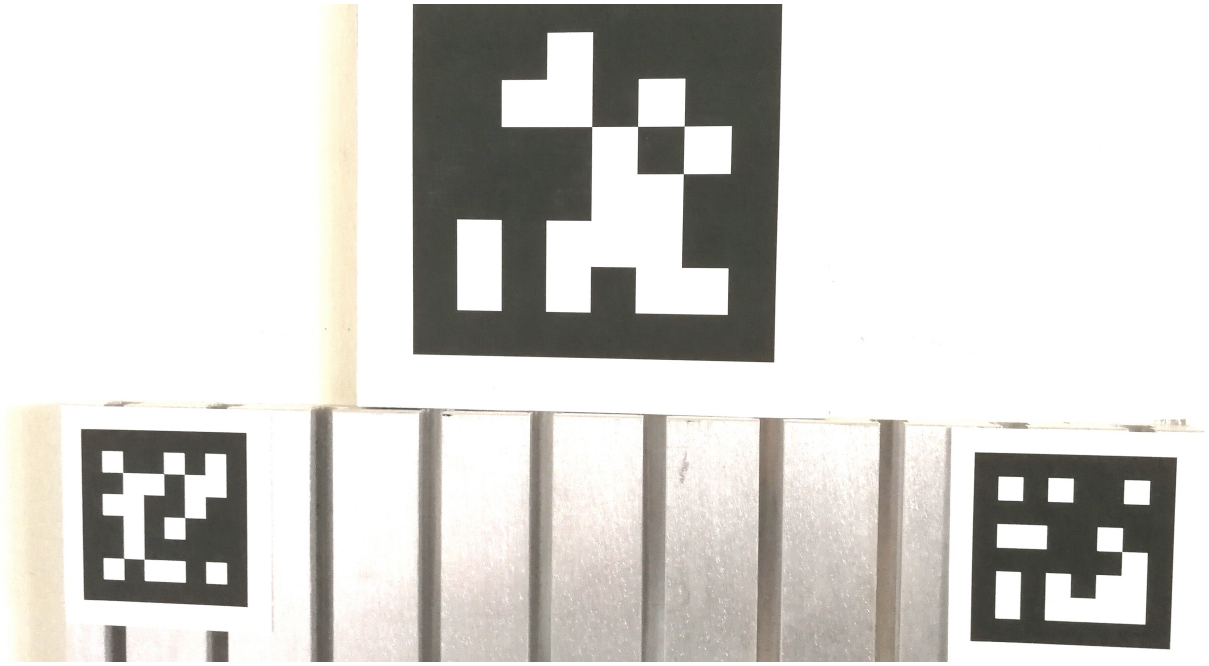


Figure 2.7: Several AprilTags in different sizes

3 State of the Art

Pouring with a robotic arm can be approached in different ways because there are multiple sources of information that can be used for generating a pouring motion. Many researchers have approached this task, only focusing on a single aspect (computational/visual fluid dynamics, force sensor data, learning algorithms, etc.) or only using simulation [21, 22, 23]. When robots are used, simple pouring tasks with mostly predefined parameters and slow movements have been demonstrated in practice [24, 25].

Multimodal systems have to be developed dealing with the complexity of real world environments to enable more advanced pouring. Optimization of data, parameters and learning algorithms are important aspects that have to be tackled within this context.

In the following sections an overview of approaches is given using different kinds of data to complete pouring tasks and some drawbacks are mentioned. Section 3.1 outlines liquid simulation which is generally slower than most other approaches. Section 3.2 describes different force sensor data methods which are difficult to transfer on to other robots. Section 3.3 presents approaches using camera sensor data which tends to be imprecise or relies on certain properties like transparent objects and colored liquids. Section 3.4 outlines the use of recorded motions which is difficult for creating a working generic solution. Finally a hybrid approach is explained in section 3.5.

3.1 Liquid Simulation

In liquid simulations the motion of the liquid is precomputed and the trajectory is adjusted to the desired liquid motion result. 3D models of the pouring and receiving container must be available for this approach. Simulation can be applied live or for the training purpose of a pouring model. Algorithms used for simulating liquid are often based on the Navier-Stokes equations [26].

[27] generates their trajectory using three constraints.

1. Collisions
2. Smoothness
3. Liquid Simulation

In the last constraint the liquids is represented by small particles and the constraint is optimized by maximizing the amount of particles falling into the glass. Computing that constraint with 1000 particles takes roughly 1 hour which makes it unfit for most real life applications. Furthermore the approach was not tested with real robots.

3.2 Force Feedback

In [22] the data of a force-torque sensor is used to generate the next joint states for a pouring trajectory. A prediction model is previously trained remotely controlled by a human (not real human motion). Retraining is needed with different shapes and the approach was only tested with tiny metallic spheres as a substitute for liquid.

Force sensor data is fed into a recurrent neural network in [28] for generating pouring trajectories. The data is collected by recording a human pouring liquids, using a custom made 3D printed adapter that connects a cup and the force sensor. The resulting trajectories are also not evaluated on a real robot.

3.3 Image Processing

[24] uses the image data of a camera sensor to measure the amount of liquid inside the receiving container in real time. Additionally the flow rate is estimated using analytical algorithms based on the pouring container shape. The desired pour angle is calculated with following inputs:

1. Amount of liquid poured
2. Flow rate
3. Current pour angle

The measurement is performing well even when sloshing occurs. An AprilTag (section 2.7), a transparent receiving container and colored liquid are necessary for the approach.

Another way of measuring liquid, without needing transparent containers or colored liquids, is presented in [29] where the liquid level is measured using only the depth information of an RGB-D camera. In order to deal with different kinds of liquids multiple solutions are implemented. The type of liquid has to be given in order for the right algorithm to be selected. Transparent liquids such as water and olive oil are more difficult to estimate because they refract the light leading to incorrect depth data, carbonated water being the most error-prone liquid. While being good for preventing overflowing, the actual amount of liquid can only be known given the shape of the container.

[30] uses the images of two different camera sensors to train a deep network to imitate movements. A key aspect is that one camera stays at a fixed position while another is moving around the executed movement. In their paper pouring is successfully imitated but without high precision.

3.4 Motion Analysis

Recording human motions is an effective first step for trajectory generation. How they are recorded is not very significant as long as the data is precise enough. A bigger challenge lies in the data processing in order to achieve a generic solution from it afterwards.

[31] proposes an algorithm to segment recorded trajectories and generate a motion, based on them. It is successfully tested on a real robot for moving a kettle towards a glass but actual pouring is not performed.

3.5 Hybrid Approach

Force sensor data and liquid simulation are both used in [32] but the results show that over longer optimization cycles the simulation by itself often performs better than the hybrid approach. The amount of liquid spilled is measured with a scale placed beneath the receiving container. Except for the pouring, everything is manually set up for each iteration. The approach consists of two phases.

1. **Calibration**
   The simulation is simplified for faster computation, in order to improve its accuracy it is calibrated first by measuring real spillage after executing a pouring motion with the robot and optimizing the simulation to match it.
2. **Optimization**
   The trajectory is played back in simulation and on the real robot and the spillage measured. Then the trajectory generating algorithm is optimized to minimize the spillage.

4 Concept

This chapter presents a concept for an adaptive pouring solution, listing its requirements in section 4.1, proposing solutions for the practical achievement of these requirements section 4.2, and exposes the limitations in section 4.3.

4.1 Requirements

The established requirements for an adaptive pouring behavior keep humans in mind as robots will need to collaborate with them more in the future. The emotional aspect has to be considered when interacting with humans, e.g., a commercial robot may not be used or bought if it is not liked. Therefore part of the concept is being predictable which makes it different from usual pouring requirements in manufacturing. A list of the basic requirements is explained below.

1. **Smooth and Expected Motions (no Trembling)**
   Trembling can lead to spilling liquid and does not look natural to humans as they usually only tremble if something is wrong. Unexpected motions can lead to fear (e.g., sudden fast motions), anger due to impatience (e.g., slow motions), surprise (e.g., motions into the opposite direction than the one expected) and even cause injuries (e.g., fast motion into unexpected direction). In order to achieve the most natural motions possible, real human motions are recorded as a reference (chapter 5).
   Limiting the joint speeds of a robot to solve the speed problem is not enough because it does not control the end effector speed: Moving multiple joints with the same speed limits into the same direction can cause the end effector to reach a fast speed while moving only one joint can result in being too slow despite identical joint speed limits. Therefore the speed of the end effector itself should be limited (does not work out of the box with MoveIt!). Further challenges regarding the prevention of unexpected motions are described in section 6.4.8.
2. **Precise and Consistent Amounts Poured**
   The requested amount of liquid to be poured should be consistently fulfilled as precisely as possible with changing parameters (section 4.2).
3. **No Spillage**
   No liquid should be spilled which is obvious but difficult to implement as the reasons for spilling are numerous and often caused by complex physical properties of water. Consequently this requirement can be subdivided into more realistic goals.
   - Not missing the glass while pouring
   - Not tilting the bottle too far
     If air can not come through the mouth of a bottle during pouring a vacuum is created and the flow is stopped for a moment. This causes liquid to flow irregularly which often leads to spilling.
4. **No Waiting Time → Fast Computation**
   If a robot takes too long to react humans tend to think it is broken as they are used to each others comparably fast reactions.
5. **Adaptive (no Retraining/-programming)**
   Humans do not expect the same intelligence from a robot as other humans but they can still differentiate between an *intelligent* and a *stupid* robot. The goal of the pouring behavior is the robot being considered as the former. Achieving the adaptation to the parameters listed in section 4.2 is a step in that direction.

4.2 Input Parameters

Changing environments lead to the need of motion adjustments to keep the pouring results consistent. In the best case the robot adapts its pouring behavior to every change that can influence the pouring outcome. In order to create a realistic concept multiple parameters representing pivotal changes affecting the pouring motion are listed and solutions for retrieving these parameter values are described in more detail. A change can also mean a request aimed at the robot directly (parameters 4. and 6.).

1. Location of bottle and glass
2. Height of bottle and glass
3. Obstacles
4. Amount of liquid to be poured
5. Amount of liquid inside bottle
    - Weight of empty bottle
    - Total weight of bottle
6. Pouring type (normal/high/slow/spout)
7. Viscosity (syrup/water)

Table 4.1: Selected changing parameters that the pouring motion should be adjusted to

1. **Location of bottle and glass**
   The `find_object_2d` package is used to identify and locate the bottle and glass at the same time. The package consists of OpenCV[1] feature detection implementations and a GUI which is used for recording and saving features. The GUI is also used for comparing the feature detection implementations and choosing the most successful one. An AprilTag (section 2.7) lets the camera locate itself in respect to the robot which is modeled in a URDF file.
2. **Height of bottle and glass**
   Identifying the height of the bottle is important for grasping it and for know the pouring end point (the `Pose` of the bottle). The height of the glass is needed to set the elevation of the pouring motion. This parameters can be identified through camera images but it is simpler to store properties of different bottles and glasses in a database and then identify them using the `find_object_2d` package mentioned in the point above.
3. **Obstacles**
   Using MoveIt! every obstacle in the robot's planning scene is considered during motion generation. Using a camera's depth sensor for recognizing unknown obstacle can also be considered but that is a complicated task to implement. The force torque sensor can also be used to recognize objects in the way but only after already colliding with them and only for the gripper link where the force torque sensor is located.
4. **Amount of liquid to be poured**
   The amount can be directly or indirectly specified by a human through any kind of interface. Either the human specifies a type of available liquid and the amount or chooses a cocktail for which each liquid and its amount is stored in a database.
5. **Amount of liquid inside bottle**
   The initial amount of liquid is important for motion generation because the angle at which liquid starts pouring out of the bottle depends on it. It is divided into two sub-parameters:

[1] Open Source Computer Vision Library

- Total weight of bottle
- Weight of empty bottle

The total weight is calculated using the force torque sensor. The weight of the empty bottle is read from the database mentioned in the point above.

6. **Pouring type (normal/high/slow/spout)**
The ability to do the same things differently makes a robot more human-like. Through this parameter the way the robot pours can be defined. With the data collected in the experiment (chapter 5) four pouring types, explained in page 45 are possible to recreate.

   Instead of manually setting the pouring type each time, a random human like behavior can be implemented. Different constraints have to be considered in that case, e.g., pouring from higher positions requires a larger amount being poured due to starting at a low point at first to avoid spillage (seen in page 48).

7. **Viscosity**
Liquids with different viscosities flow with different speeds and therefore influence the pouring time. A viscosity property can be added to the database mentioned in the second point as a property of each bottle.

   Another idea is automated the viscosity recognition using the force torque sensor by identifying the change patterns of the sensor data for different viscosities. This can be done when shaking the bottle before pouring or during pouring which is more risky. The method is unlikely to succeed as the data needs to be very precise to detect such changes.

4.3 Limitations

Following points are not considered in the concept.

- **Filling capacity of the glass**
The volume of the glass is is difficult to obtain without having an exact 3D model of the glass. The height of the glass can be used to estimate a maximal filling capacity instead.
- **Initial amount of liquid in glass**
This parameter is not seen as important because glasses are usually filled starting empty. Nevertheless it is an interesting parameter with following ways to obtain it.
  - Measuring the current liquid level inside of the glass requires a camera viewing the inside of the glass. Knowing the liquid level does not necessarily lead to a correct estimation of the amount as it depends on the glass' volume which is unknown.
  - If the glass' empty weight is known, the current amount of liquid can be measured using a force torque sensor or a scale.
- **Different pouring and receiving containers**
Only regular bottles and glasses are used which ignores other popular pouring containers like cans, jugs or decanters. Since grasping these objects is difficult and bottles and glasses are the most common pouring containers the height parameter is considered enough for different shapes.
- **Liquid simulation**
  - Computation time is relatively high
  - Implementing the algorithm is time costly
  - Exact 3D models of containers are needed

5 Recording Human Pouring Motions

This chapter explains the laboratory experiment conducted for recording human pouring motions. The motions are recorded in order to find out more about them, and to test if the robot can imitate them successfully.

Section 5.1 lists and explains the tools used for executing the experiment. Section 5.2 describes the preparation steps and section 5.3 the actual recording process. Lastly section 5.4 goes through the follow-up steps.

5.1 Experimental Setup

For recording human trajectories a bottle is tracked using a marker-based motion capture system (PhaseSpace Impulse X2E [33]). The system consists of eight cameras that can track the position of LED markers. Each marker is glowing at a unique frequency rate allowing it to be identified by the system. PhaseSpace's patent describes the functionality in detail [34]. The data capture rate is set to 240 Hz (frames/sec) and it does not need filtering.

Apart from the tracking system, a cage frame for the cameras and the eight LED markers, following things are used for the pouring experiment, marked in fig. 5.1:

1. USB scale
2. Two bottles (tracking, refilling)
3. Funnel (avoids spilling when refilling the bottle and emptying the glass)
4. Glass (marked with 3 stripes for different configurations, seen in table 5.1)
5. Container (for catching drips and spillage)

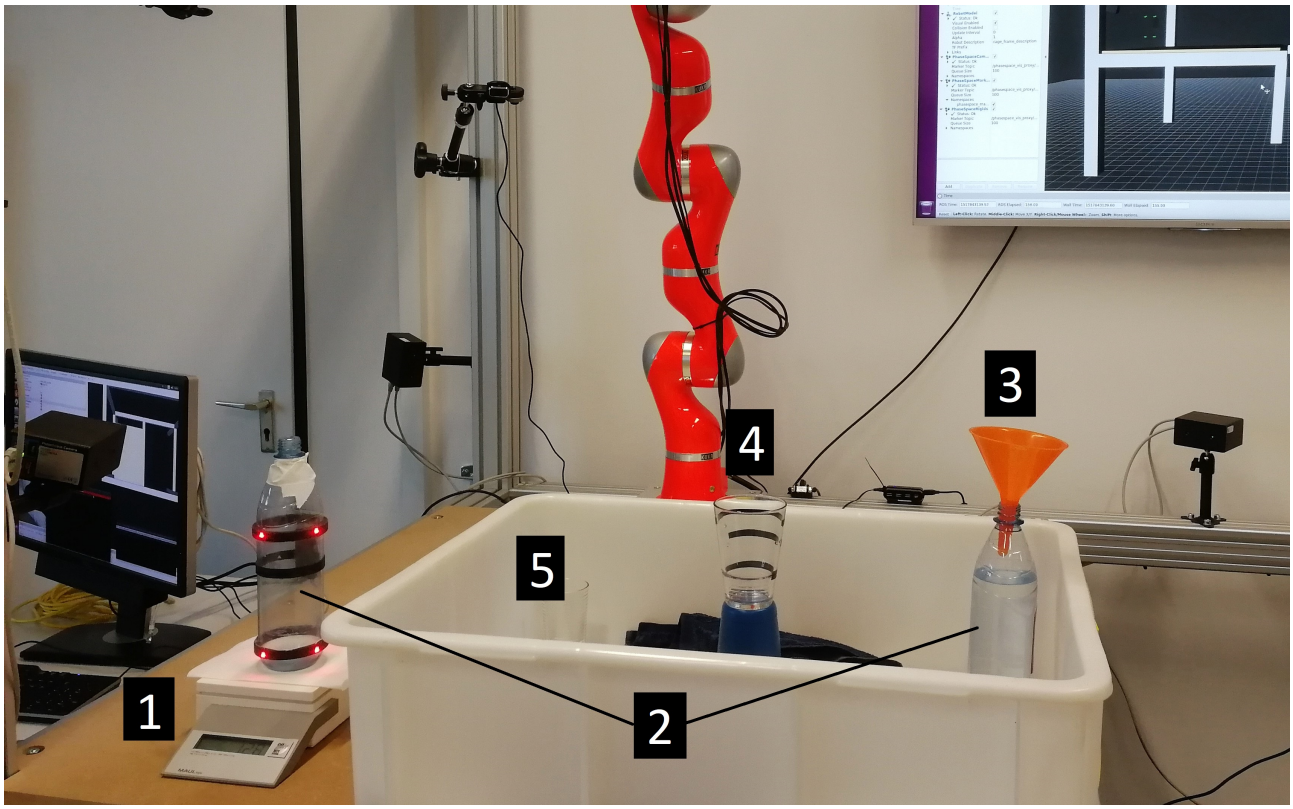


Figure 5.1: Picture of the experiment setup with marked components:
Scale (1), bottles (2), funnel (3), glass (4) and container (5)

The eight LED markers are fixed to the tracked bottle by modeling and 3D printing a bottle fitting plastic case including a battery compartment that allows easy replacement. The position of the center point on the top of the bottle (called bottle mouth in the following) relative to the markers is previously measured manually and automatically published during recording. Only three of the eight markers are needed to be recognized at all times in order to locate the exact pose of the bottle mouth.

5.2 Preparation

Before recording the pouring motion of humans a few preparation steps are taken:

- The container is fixed to the table avoiding a displacement of the glass inside.
- A cup is placed upside down inside the container and fixed with tape. Its purpose is to elevate the glass so that the view on the markers of the bottle can not be blocked by the high container edges.
- The position of the glass is determined by holding the bottle mouth to the bottom position of the glass and recording its position as seen in fig. 5.2.
- The empty bottle is placed on the scale to determine its empty weight.

5.3 Recording

When the whole setup (motion capture system and all relevant ROS nodes) is running, a script is executed that starts recording a bag and saves it with a given name (listing 5.1).

Listing 5.1: Format of bag name for documentation

```
<configuration>/<test person>/<number of emptied bottles>
```

The main topics recorded in each bag contain the following information:

- Bottle position and rotation
- Weight on the scale
- Camera image (for monitoring)
- Transformation tree (transformations between the frames on the right in fig. 5.2)

Following steps are then repeated as long as desired during recording:

1. Refill pouring bottle if empty with the second bottle through the funnel
2. Pour water into the glass according to the current experiment configuration (table 5.1)
3. Move bottle outside of the cage if a problem occurred (e.g., spilling)
4. Place bottle back onto the scale
5. Pour water from glass into refill-bottle through funnel

Configuration	Pour Marker	Spout	Slow	High
1	1			
2	2			
3	3			
4	2		x	
5	3		x	
6	3			x
7	1	x		
8	2	x		
9	2	x		x

Table 5.1: Recorded configurations: The pour marker is one of the three black stripes on the glass (No. 4 in fig. 5.1) that the liquid is poured up to each time. Spout indicates that a bottle spout is attached to the bottle during pouring. Slow means that the movement is done as slowly as possible. High means that the liquid is poured from as far away as possible from the glass.

5.4 Follow-up Processing

The glass' coordinates are recorded in a separate bag by placing the bottle on top of the upside down turned cup placed underneath it (fig. 5.2). Afterwards the transformations between the world, the table and the glass are retrieved using the `tf` package while playing back the recorded bag.

The glass is not marked in the experiment because attaching markers to a fixed object is too costly. The position could not be measured and set beforehand because the exact placement of the glass was not clear until the beginning of the experiment. The glass perfectly fits onto the upside down turned cup that elevates it (not planned) which is why it does not need to be fixed with tape. This would have slowed down the experiment every time the glass is picked up to be emptied.

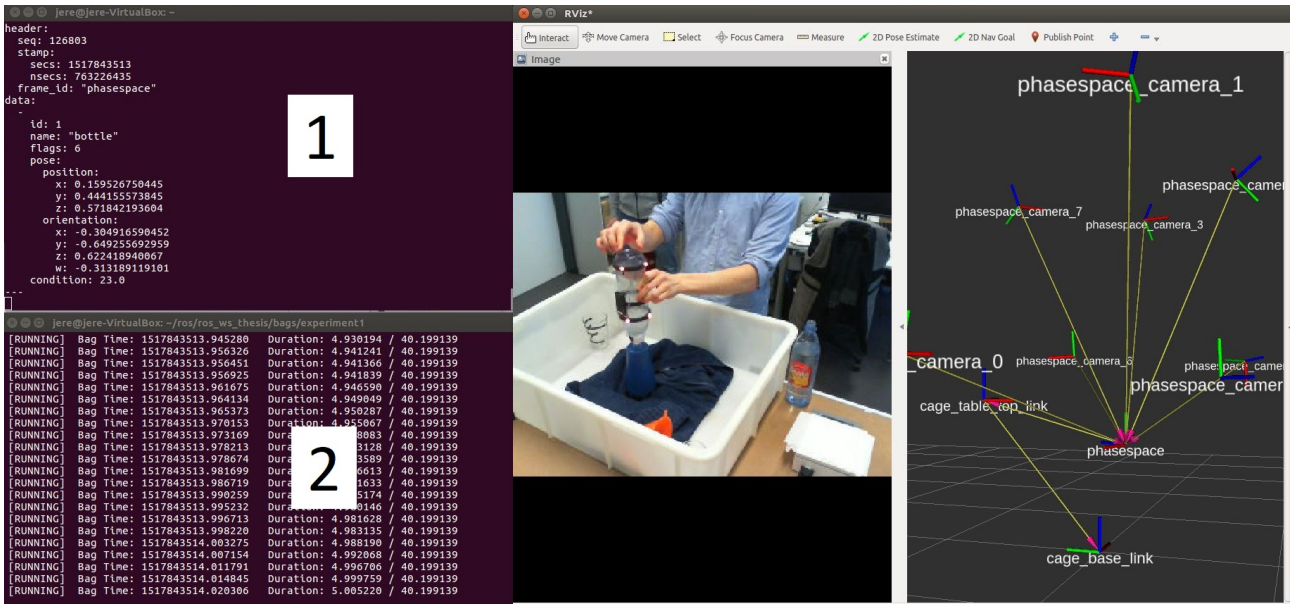


Figure 5.2: Retrieving the glass pose by playing back the recorded bag with `rostopic echo /phasespace_ros/rigids` (2) and looking at the motions coordinates by echoing the bottle topic with `rosbag play setup_glass.bag` (1)

6 Implementation

This chapter describes the implementation and functionality of the prototype made for analyzing the recorded human motions in chapter 5 and replaying them on the robot in detail.

The implementation is not focused on in the concept presented in chapter 4 which deals with the whole pouring behavior more broadly.

An iterative process of modularization is performed during the course of development. Small nodes are developed first consisting of only a few needed functions. Variables and methods are wrapped into classes later on. Messages are added to reuse the nodes' output as input for other nodes. Each class is then split into a header and an implementation file, and included into other classes where it is used. This leads to simpler reuse of code by not having to communicate through messages anymore. On the other side changes of method signatures have to be modified in both header and implementation file. A coordination node including most classes is implemented, and services and actions are added. For better control of the input parameters a web user interface (WUI) is created. Finally the `tams_pour` package is used in the `tams_ur5_bartender` package to execute the entire pouring behavior while minimizing the code changes in the latter.

Section 6.1 explains the automated extraction of recorded pouring motions. How the motions can be analyzed is described in section 6.2. Section 6.3 introduces the visualization of the motions. The class for traversing the motion with the robot is explained in section 6.4. Helper methods outsourced to a separate class are introduced in section 6.5. How the original motion data is smoothed is described in section 6.6. Section 6.7 lists the services and actions created, while section 6.8 describes the coordinator that connects many of the components. Finally section 6.9 presents the implemented WUI functionalities.

6.1 Extractor

The extractor is the class used for automatic extraction of the recorded motions. While recording the pouring motions, no notes have to be made when a single pouring motion is completed because this is done by the extractor automatically. It saves time because a single person can do it without the need of manually pressing record/stop or having to mark or delete failed samples after each try. Time consuming cutting of videos to split all motions is also not needed. To mark failed samples the bottle only has to be moved outside of the pouring area. More data can be recorded and analyzed in less time this way. The extractor is especially helpful for experiments with the goal of obtaining as much data as possible (e.g., for machine learning).

The extractor has a set of input parameters that can be changed for flexible usage.

1. Folder path of imported bags
2. Array of imported bag filenames
3. File path of exported bag
4. File path of exported comma-separated values (CSV) file
5. Weight of bottle
6. Pouring area x-axis limit
7. Pouring y-axis toggle
8. Minimal tilt angle
9. Maximal weight difference
10. Minimal weight repetition
11. Show whole bag after extraction

After extraction the option `Show whole bag after extraction` can be set to either display each motion in Rviz one by one or all motions of each bag at the same time.

A recording setup can change in following ways without needing to reprogram the class:

- The pouring and receiving container can be changed to any shape.
  The `minimal tilt angle` may have to be adjusted.
- All components can be placed anywhere.
  The `pouring x- and y-axis` parameters may have to be adjusted.

6.1.1 Bag Import and Processing

The imported bags are traversed and each pouring motion is extracted. Each bag's extracted motions are saved in a list which is then again saved in a list of bags as visualized below. After traversing a bag the list of motions is added to the list of bags, emptied, and reused for the next bag.

List of bags
- List of motions in bag1
  - Motion1
  - Motion2
  - ...
- List of motions in bag2
- ...

While going through the data of each bag, two topics are read:

- **Bottle (`PoseStamped`)**
  The bottle is transformed from its original frame (world) to the table frame by multiplying the transformation from world to table with it:

$$Bottle_{Table} = {}^{World}Transformation_{Table} \times Bottle_{World}$$

  The table and glass frame are added to a URDF with their known relative position to each other. This way the last transformation to map the bottle to the glass frame can be applied. The final bottle transformation is converted back to a `PoseStamped` and the timestamp is subtracted by the first timestamp in the current motion. This ensures that each motion starts at zero seconds. Finally, the `PoseStamped` is added to the motion.

- **Scale (current weight)**
  Each scale value is initially compared to its previous value. If the difference is greater than the `maximal weight difference` parameter it is counted as a change of weight, otherwise as an unchanged weight. If the weight remains unchanged for the number of times given in parameter `minimal weight repetition` it counts as a `stable weight`. The last three `stable weight` values are stored in variables and compared in order to detect the end of a pouring motion as shown in fig. 6.1.

  When the end of a motion is reached, the amount poured is computed by subtracting the current `stable weight` from the oldest `stable weight` (two steps back). The initial amount is computed by subtracting the weight of the bottle from the oldest `stable weight`.

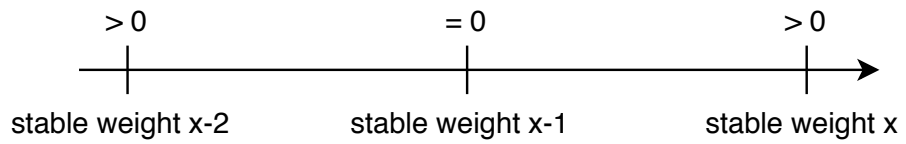


Figure 6.1: Condition on which a pouring motion is extracted: An `stable weight` with the value zero in between two `stable weight` values above zero

After the end of a motion is identified, a trajectory message (listing 2.7) is created and all properties are filled. The `valid` property is set by evaluating all motion failure checks in section 6.1.2 and if at least one of the checks fails it is set to `false` (invalid). The pouring configuration number (table 5.1) is extracted from the current bag's filename, and further information, mapped to the number (`high`, `slow` and `spout`), is set. After being copied to the list of motions, the `PoseStamped` objects are removed and failed check variables are reset.

6.1.2 Failed Motion Checks

Each time a value from a bag is read, checks are done to verify the current motion's correctness.

- **Motion Repeat Check**
  Every time the distance between the bottle and the glass on the y-axis, rises above or falls below the `pouring y-axis toggle` value, a counter increases (algorithm 1). In order to avoid false counts due to jitter at the points near the `pouring y-axis toggle` value, each compared point must be a specified distance away from the previous point. When the counter is above two (= forth, back, forth, ...), the motion is set to invalid because it is repeated. This means, that the scale value based extraction failed to extract a motion before the next one started.

if *(previousPointY* $<$ *crossDist AND currentPointY* $>$ *crossDist)*
OR (previousPointY $>$ *crossDist AND currentPointY* $<$ *crossDist)* **then**
  crossedTheLine = crossedTheLine + 1
end

Algorithm 1: Motion Repeat Check (`crossDist` = `pouring y-axis toggle`)

- **Boundary Check**
  The x-value of the bottle is compared to the `border of pouring area` parameter. If larger, the motion is marked as failed because this means that bottle is outside of the pouring area which is agreed upon being the indicator for spilling or other failures during pouring.

If a motion is not marked as invalid by the boundary check, two further checks are applied to the bottle's `PoseStamped`:

- **Tilt Check**
  The tilt angle is compared to the `minimal tilt angle` parameter. If it is larger, the motion is considered correct in this aspect. If not a single `PoseStamped` in the motion is larger, the motion is marked as failed.
- **Poured Amount Check**
  To filter out other falsely identified motions the amount poured is checked. If its value is not above 0 ml the motion is marked as failed. The exact cause of this is not known.

6.1.3 Output

All extracted trajectory messages are written to one bag in the end. Analysis can be performed faster on one bag because it takes less time to open one bag, than multiple bags. For processing the data with other, more standardized tools, the motions are saved as CSV files. A single CSV file is created for each motion in order to pick out desired ones more easily, and not needing indicators for the end of a motion. The timestamps and the poses are saved inside of the CSV file, while the other information is saved in the filename itself. The format is `init_X_pour_X_config_X`, which describes the initial amount and the poured amount followed by their rounded values (ml), and the recording configuration (table 5.1).

Lastly all motions can be traversed in Rviz one by one for direct verification. Failed motions are marked black, poses outside of the pouring area in red. Valid motions can be displayed in different color codings from red, green to blue depending on the relative time passed or tilt angle at each pose.

6.2 Analyzer

The `analyzer` class filters the motions extracted by the `extractor`, using input parameters and displays them in Rviz. It is meant to enable comparing or searching for specific motions. The logic of this node was first implemented in the `extractor` but decoupled later because it is too time consuming to go through the extraction process each time before analyzing the motions. The implementation of the actual visualization in Rviz is also done here at first but is decoupled to the visualizer class (section 6.3) later.

The `analyzer` receives multiple input parameters.

1. File path of imported bag
2. List of motion ids
3. Minimal tilt angle
4. Minimal amount poured
5. Maximal amount poured
6. Minimal initial amount
7. Maximal initial amount

The first action of the `analyzer` is to import all trajectory messages from the given bag into an internal list. Next there are two possibilities:

- Specific motion filtering
  A list of motion ids is given and displayed if available.
- Property filtering
  All motions are searched for given property values and each motion that fits is displayed

6.3 Visualizer

A visualizer node is implemented because all classes make use of visualization for displaying their results. Its main methods are used for publishing motions in a format that can be displayed in Rviz.

While being processed in the other classes, motions occur in different types such as `Pose` and `PoseStamped` lists or trajectory messages. Converting them to a different type to match a publishing method's input parameter type each time is not efficient, therefore multiple publishing methods for different motion types are provided. One central method iterates over the

motion's `Pose` objects, adds colors based on their values, creates a marker object displayable in Rviz and publishes it. In another method a `moveit_visual_tools` class is used for publishing motions. It is easier to use but its smallest marker size is still too large sometimes to clearly see the motion line and the color can not change within the motion line. Furthermore an arrow visualization of the motion is implemented in order to visualize the orientation of each pose more clearly. The different views are helpful when trying to find explanations for why a motion plan solution is not always found in some cases and also for comparing multiple motions to each other. A comparison of the three basic visualizing methods is shown in fig. 6.2.

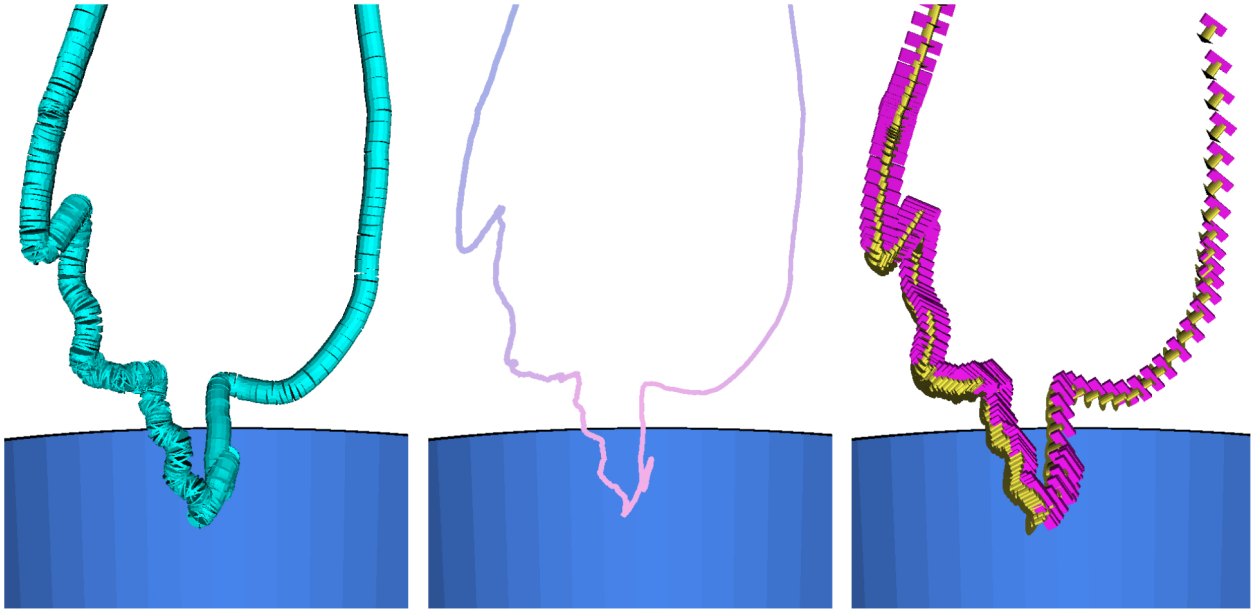


Figure 6.2: Using the visual tools provided by MoveIt! (left) trajectories can be visualized easily but not as precisely and detailed as using a publisher with custom markers (middle and right).The visualization in the middle is created using a `LINE_STRIP` marker, the blue color turns pink as the tilt angle increases. In the trajectory on the right every third point is replaced by an yellow arrow and a purple box for indicating the arrow's rotation.

6.4 Pourer

The pourer is the most complex of all classes in this thesis because several computations have to be made and different parameters adjusted in order to successfully move the robotic arm.

The entry point takes in a trajectory message and keeps on using it instead of only using a list of poses. This ensures an easy adjustment if methods are added that need to use other information from the trajectory message (e.g., `valid`, `slow`,...). The motion is processed in different steps before being executed by the arm. Each step is visualized in Rviz for monitoring the process.

Through the pouring service properties presented in section 6.7 the `pourer` can be influenced.

6.4.1 Transformation (bottle to gripper)

The transformation from the gripper target (middle of the bottle) to the bottle mouth (${}^{E}T_{B}$) is a displacement of half of the bottle height on the z-axis. The transformation from the glass to

the bottle mouth (${}^{G}T_{B}$) is known, and the transformation from the glass to the gripper (${}^{G}T_{E}$) is needed. The intuitive thought is dividing the known transformations but matrices can not be divided directly. Instead they have to be multiplied by the inverse as shown below.

$$ {}^{G}T_{E} \times {}^{E}T_{B} = {}^{G}T_{B} $$

$$ {}^{G}T_{E} = {}^{G}T_{B} \times {}^{E}T_{B}^{-1} $$

6.4.2 Rotation of Motion

The recorded motion is rotated towards the robot in order to be reached as quickly as possible from the current gripper position (fig. 6.3). If the shortest path is blocked or not traversable because no IK solution is found, the motion is rotated by a specified number of degrees to the left and right from the shortest path until a solution is found or a full circle around the glass is made (fig. 6.4). The alternating rotation is done in order to find a possible candidate quicker because it is more likely to be near the shortest path. The alternation is achieved with following formula, the direction alternating between -1 and 1 on each iteration and the angle step being a value in degrees, that can be freely set:

$$ Rotation_{Current} = direction \times iteration \times angleStep $$

A motion is rotated as follows: Each `PosesStamped` object of the motion is rotated around the z-axis of the glass by the same amount. Since the `PosesStamped` objects are already in the glass frame the z-rotation matrix (2.3) can be directly applied to each one. The angle by which the poses are rotated to be in line with the gripper is computed with following method, E being the gripper's `Pose` and P the first `Pose` of the motion:

$$ Rotation_{TowardsGripper} = \arctan 2(E_y, E_x) - \arctan 2(P_y, P_x) $$

The rotation of E is subtracted by the rotation of P. This way the relative angle is known by which the motion is rotated to be in line with the gripper.

6.4.3 IK Solutions

Before playing back a recorded motion, the robot has to be able to reach the first `Pose`. This is tested by using an analytical IK solver specifically made for the used robotic arm, called `UR5 Kinematics Plugin` from the package `ur_kinematics`. Finding a single solution is not enough in some cases as subsequent waypoints may not be able to reach from the initial joint positions. To find more possible solutions the solver is executed multiple times. The solver returns the best solution based on the current state of the robot which is why the initial state of the robot is set to a random state after the first solution is computed in order to receive different possible solutions. Up to eight solutions can be received depending on the target. The solutions are at risk of not being reachable from the current state of the robot or causing self collisions but the planning method of MoveIt! checks this automatically afterwards.

6.4.4 Joint Trajectory

For each IK solution found CCP is executed for the given motion. The resulting joint trajectory rarely fully traverses the original motion but usually planning success of around 90% is sufficient.

Figure 6.3: Trajectory rotation towards gripper and orientation adjustment for gripper pose (blue and red colored trajectories) transformation from top-down perspective

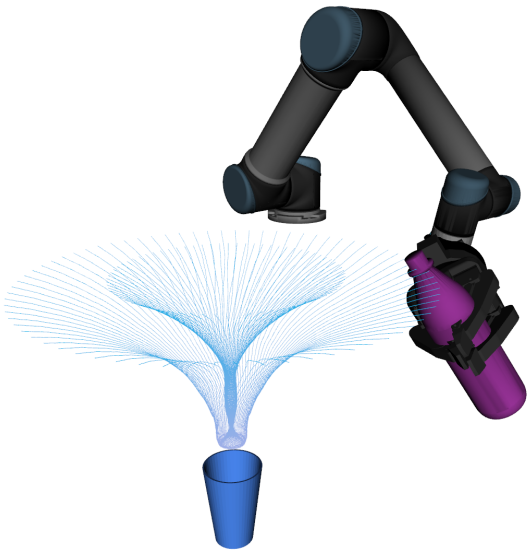


Figure 6.4: Trajectory rotation until a solution is found, the step size is adjustable

The `minimal plan completion` parameter is used for deciding when the result is interpreted as successful.

6.4.5 Velocity Adjustment

The velocities of all joints are adjusted after the computation of the joint trajectory because CCP does not take into account the original timestamps of the motion. Generally the velocity value is calculated dividing the distance between the previous and the next point by the time passed between them which results in the average velocity that the current point is set to. The first and last point's velocity is set to zero.

$$v_i = \frac{(p_{i+1} - p_{i-1})}{(t_{i+1} - t_{i-1})}$$

6.4.6 Integration of Reflexxes

The Reflexxes library (section 2.6) has to be used for smoothing the movement of the robot, because the velocities computed after applying the original timestamps in section 6.4.5 do not result in smooth motions. A method that receives a joint trajectory and returns a smoothed one is implemented. Given the time between each point and the speed limits of the robot it calls the library's method repeatedly and stores the output in a new joint trajectory until the last point is reached. The pseudo code in algorithm 2 depicts the logic of the implemented method.

```
Function AdjustTrajectory(trajectory, maxVel, maxAcc, timeToNextPoint):
    outputTrajectory = empty
    i = 0
    start = trajectory[i]
    add start to outputTrajectory
    while i < trajectory.length do
        goal = trajectory[i+1]
        limits = ⟨maxVelocity, maxAcceleration⟩
        result = empty
        while goal != result do
            result = Reflexxes.computeNextPoint(start, goal, limits, timeToNextPoint)
            add result to outputTrajectory
            start = result
        end
    end
    return outputTrajectory
```

Algorithm 2: Method for smoothing the robot's movement using the library Reflexxes

6.4.7 Pouring Process Sequence

The following steps are repeated until a complete pouring solution is found and executed:

1. Rotating
2. Computing IK solutions for first trajectory point
3. Removing gripper constraints
4. Planning pouring path (`pp`) with CCP
5. Applying gripper constraints
6. Planning path (`fp`) to first pouring point
7. Planning path (`lp`) from last pouring point to initial `Pose`
8. Removing gripper constraints (in order to execute `pp`)
9. Executing plan `fp`, `pp` and `lp` in that order

6.4.8 Constraints

Constraining the robot's movements is needed for achieving elegant pouring motions, because otherwise the resulting motion plans are often unexpected. The plan found for moving to the beginning of the pouring motion is often counter intuitive for humans, and the bottle is tilted too far on occasion, which is why additional constraints are needed (page 33). Each link of a robot can have several constraints. When grasping a bottle and moving it towards the glass and back constraining the gripper is of highest importance because unconstrained motions could lead into rotating the bottle upside down. During the actual pouring motion these constraints have to be removed again.

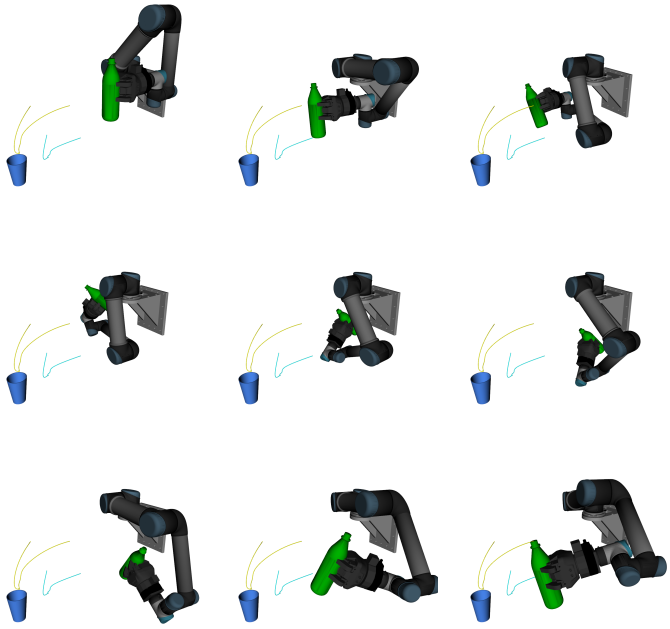
Two types of constraints are created for the gripper.

1. **Orientation**
   Orientation constraints are added limiting the orientation of the gripper. It is constrained by setting maximal rotation values for the x- and y-axis in respect to the table frame to prevent spilling. But even then some motions are unnatural (fig. 6.5b) which is why another constraint is created.

2. Position

In order to prevent the gripper from moving below the table during a motion a position constraint is added. An area that spans from the table top to approximately 1 meter above is set as the constraint. This decreases the solutions found dramatically, e.g. no solution is found from the start position shown in fig. 6.5 anymore. More constraints are needed to prevent the robot from moving too far left and right and they still do not result in reasonable motions each time. Problems during the practical implementation of constraints in MoveIt! are described in section 8.2.1.1.

(a) Without contraints the bottle is sometimes turned upside down.



(b) Orientation constraints on x- and y-axis can still lead to unexpected motions.

Figure 6.5: The path of the robot arm from its starting point to the first point of the pouring motion in sequence (top left to bottom right)

6.5 Helper

Not only the visualization is used in different parts of the code but also other functionalities are needed recurringly. The `helper` class is created to lower code redundancy by providing methods that are used in more than one class.

- **Computing Tilt Angle**
  Computing the tilt angle of the bottle is done by calculating the dot product of the bottle orientation vector and a vector that is pointing straight upwards (unit vector in the direction of the z-axis) and inserting the result into arccos as seen below.

$$\vec{v}_{Orientation} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \qquad \vec{k} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \qquad Tilt = \arccos(\vec{v} \cdot \vec{k})$$

- **Filters**
  Multiple filter methods are implemented for experimenting with the motions.
  - Tilt
    The tilt filter method removes all poses that are not tilted more than a specified angle.
  - Distance
    The distance filter removes all points that are farther away from the glass (distance from the glass is $\sqrt{B_x^2 + B_y^2 + B_z^2}$, B being the bottle point) than the value given.
  - Point skip
    The point skip filter only lets every X point through. The value $X = 1$ does therefore not filter the trajectory. Values below 1 are not accepted.
- **Conversions**
  - `PoseStamped` list to `Pose`
  - `Pose` and a time value to `PoseStamped`
- **Velocity Computation**
  The calculation of velocities given a `PoseStamped` list is implemented here.
- **CSV Creation**
  A trajectory message is turned into a CSV file after processing it adding more information like the tilt angle and the velocity.
- **Data Fitting Service Call**
  This method calls the data fitting service (section 6.6) for smoothing motions.
- **End Effector Positions**
  The positions of the gripper are computed from a joint trajectory and used for comparing them with original motion data.

6.6 Kernel-Based Motions Representation

Human motions usually seem smooth but this is not verified by the recorded motion data, which can be explained by the tiny corrections the muscles in the arm make to ensure a smooth motion. When zooming in on such a motion the jitter caused by these tiny corrections becomes visible (fig. 6.6). This is problematic for the robot as it leads to low CCP results, trembles, moves too slowly and looks unnatural. A Python3 script is used to apply data fitting on the motions for smoothing. The intensity of smoothing can be controlled by setting a kernel count. For each kernel a Gaussian function is generated. The functions are distributed over the number of data points. For each kernel a value is computed that is multiplied by its corresponding function output to achieve the best fit for the incoming data. The resulting motion is the output of all functions summed up at each point (fig. 6.7, page 36). The key is finding the amount of kernels that lead to a smooth motion while resulting in the expected amount poured and still being traversable by the robot in the end. Difficulties during the integration of the Python3 script are described in section 8.2.1.5.

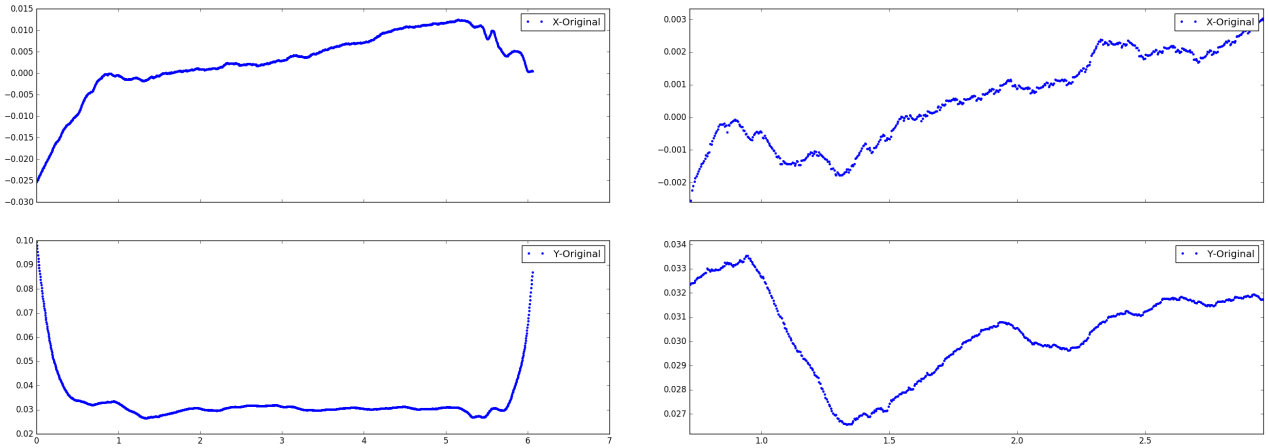


Figure 6.6: The original motion on the x- and y-axis are shown on both sides, zoomed in on the right. The jitter is visible and is not caused by noise in sensor data but by the actual movement which is usually not visible to a human eye.

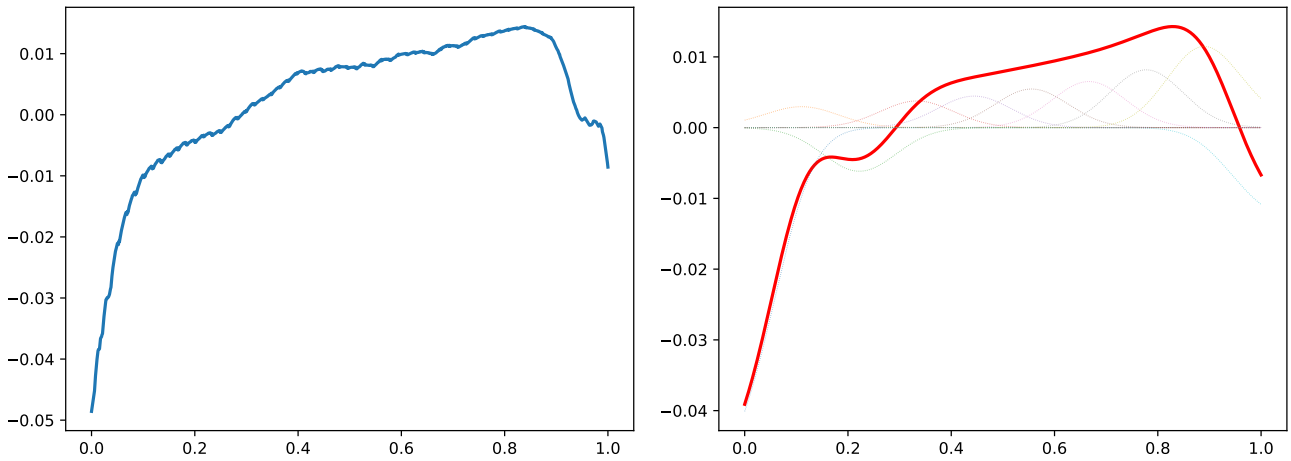
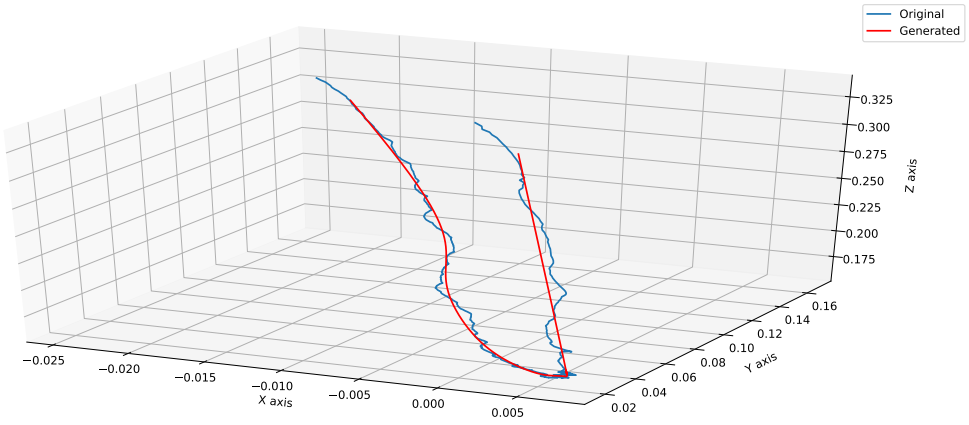
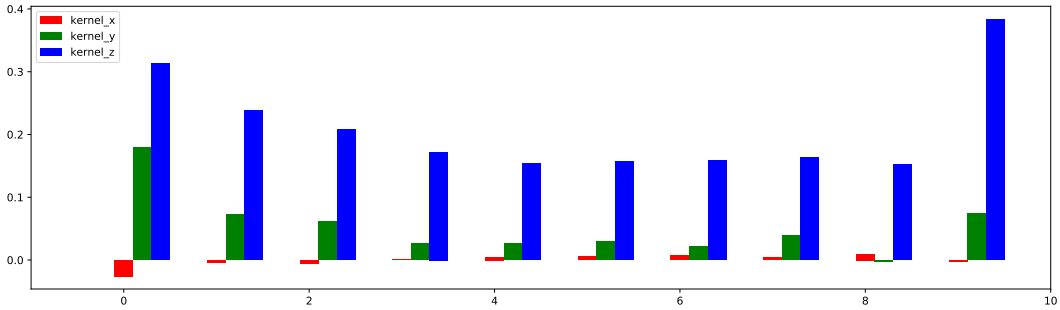


Figure 6.7: X-axis value comparison: Original motion (left) and kernel-based smooth motion (right) including the kernel functions (10) in the background



(a) An entire motion (blue) with the smoothed x, y, z values on top (red)



(b) Kernel values for all functions of x-(red), y-(green) and z-axis (blue)

Figure 6.8: 6.8a shows original and smoothed data points (blue and red) using 10 kernels. In 6.8b the corresponding kernel values for the smoothed motion in are seen.

6.7 Services and Actions

For communicating between different nodes, several services and an action are created. They are mostly used to communicate between components programmed in different programming languages, because these can not use each others methods easily otherwise. In this thesis JavaScript is communicating with C++, and C++ with Python, through the created actions and services.

Listing 6.1: GripperControl.srv

```
bool close
---
bool success
```

A service that opens or closes the gripper is created for quick testing. It is called in the WUI (section 6.9) and is implemented in the trajectory server (section 6.8) which calls the corresponding method in the pourer to open or close the gripper.

Listing 6.2: Smooth.srv

```
int32 kernels
geometry_msgs/Pose[] inputPoses
---
geometry_msgs/Pose[] outputPoses
```

A Python2 node implements a method for the smoothing service that calls the Python3 script (section 6.6). The Python3 script can not implement the service directly because ROS is not compatible with Python3 as stated by one of its lead developers in [35].

All properties inside of a `Pose` are smoothed individually. In order to obtain valid orientation values the quaternion is converted to x-, y-, z-rotation angles before smoothing, and converted back using the `tf` package.

Listing 6.3: Pour.srv

```
int32 id
float64[] parameters
---
bool success
```

The first pouring service is created with only the motion id parameter first but is extended for testing different filters. The property `parameters` enables passing any number of inputs without changing the service definition each time which is useful for testing which parameters are reasonable.

The following values are finally passed through:

- Maximal acceleration
- Maximal velocity
- Maximal distance to glass
- Minimal tilt angle
- Minimal plan completion
- Point skip between

The speed limits are passed to the Reflexxes method for testing. The other parameters are used for filtering the motion and test the best results.

Listing 6.4: PourHumanly.srv

```
geometry_msgs/Pose glassPose
float64 pouredAmount
float64 initialAmount
---
bool success
```

The second pouring service is used for the final step connecting the pourer class to the `tams_ur5_bartender` package.

Listing 6.5: ShowTrajectory.action

```
int32[] trajectory_ids
int32[] configurationFilters
bool filter
int32 pour_amount_max
int32 initial_amount_max
int32 pour_amount_min
int32 initial_amount_min
int32 min_angle
int32 skip
float64 max_distance
---
bool success
---
string task_state
```

The `ShowTrajectory` action parameters allow for two options. filtering trajectory messages based on their properties or directly by their id (position index in the list of trajectory messages) which is the functionality the `analyzer` implements (section 6.2). Information about the filtered objects is passed to the `task_state` in order to display a list of filtered motions for the user to select in the WUI.

6.8 Trajectory Server

Most of the methods of the trajectory server node implement services and actions. It is called server because it is running constantly allowing the actions and services being called multiple times without having to restart anything. The main purpose of its methods is to import the parameters given through the service or action and calling the corresponding methods of other classes. The method for pouring used for the `tams_ur5_bartender` package also filters all extracted trajectory messages in order to select the one matching the given pouring and initial amount.

6.9 Web-based User Interface

For analyzing the recorded motions a WUI is created using hypertext markup language (HTML) and JavaScript (fig. 6.9). It consists of multiple input fields and buttons for each of the services' parameters, buttons for executing a service or action and a selectable list of trajectory message information. The open source toolkit Bootstrap [36] is used to simplify the designing process and improving responsiveness when the size of the browser window is changed. Roslibjs is used for communicating with ROS through a bridge (section 2.3.9).

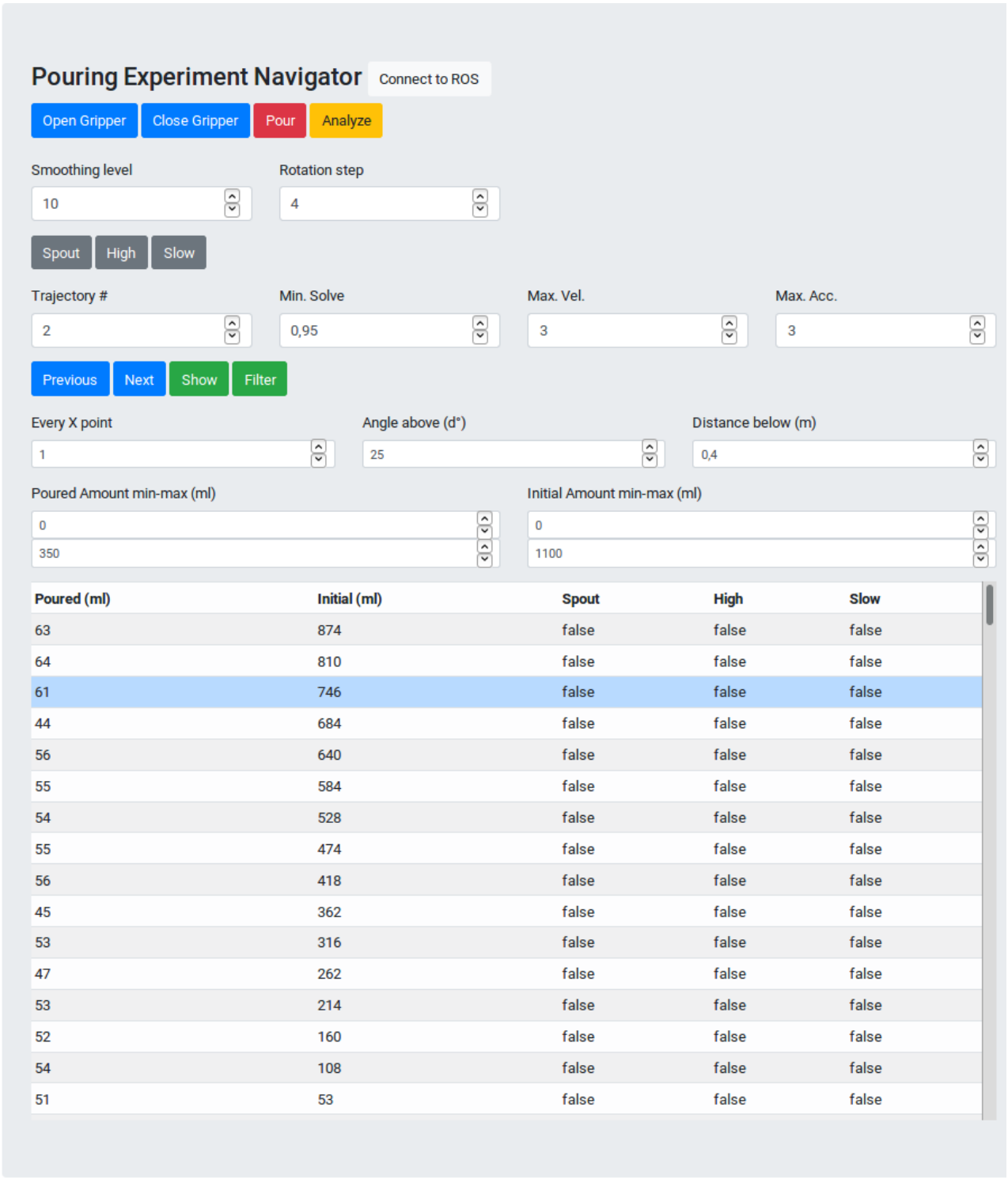


Figure 6.9: HTML interface for filtering, analyzing and traversing recorded motions, including manually optimized default values

7 Analysis

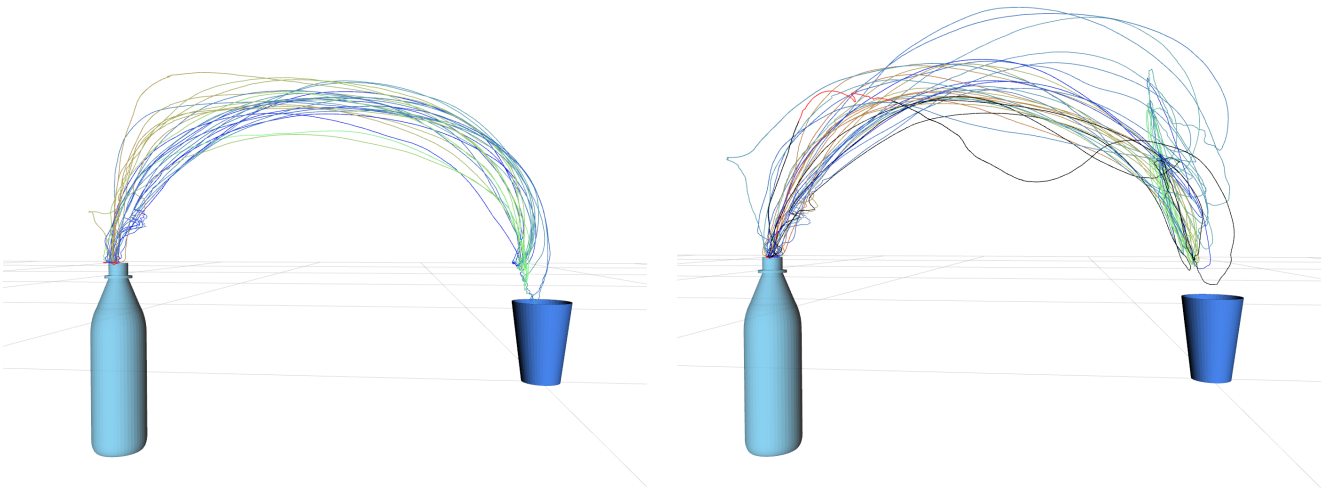
This chapter presents the data retrieved from the experiment in chapter 5 and the implementation in chapter 6, and discusses similarities and differences.

First, the results of the `extractor` class are presented in section 7.1.1. Next, in section 7.2 the recorded human motions are analyzed and shown by using the WUI to filter them with the `analyzer` class (section 6.2). Lastly, section 7.3 discusses the results of playing the motions back on the robotic arm.

Configuration numbers are frequently mentioned in this chapter; the corresponding table is found at page 23.

7.1 Extraction Results

A few of the extracted bags are presented in fig. 7.1. Invalid motions are colored in black with red parts indicating the overstepping of the specified `pouring area x-axis limit` (section 6.1). The motion is heat colored (red, green, blue) over its duration. The tiny amount of red color at the mouth of the bottle indicates, that a large amount of time has passed before the actual movement begins. This is a problem when comparing the positions over time across different and single motions, because the transitions are barely distinguishable. The `analyzer` can solve this problem by trimming the motions and resetting the time.



(a) Bag: 03, 14 valid extracted motions, Configuration: 02

(b) Bag: 09, 15 valid extracted motions, Configuration: 06 (extra high), one invalid motion clearly visible

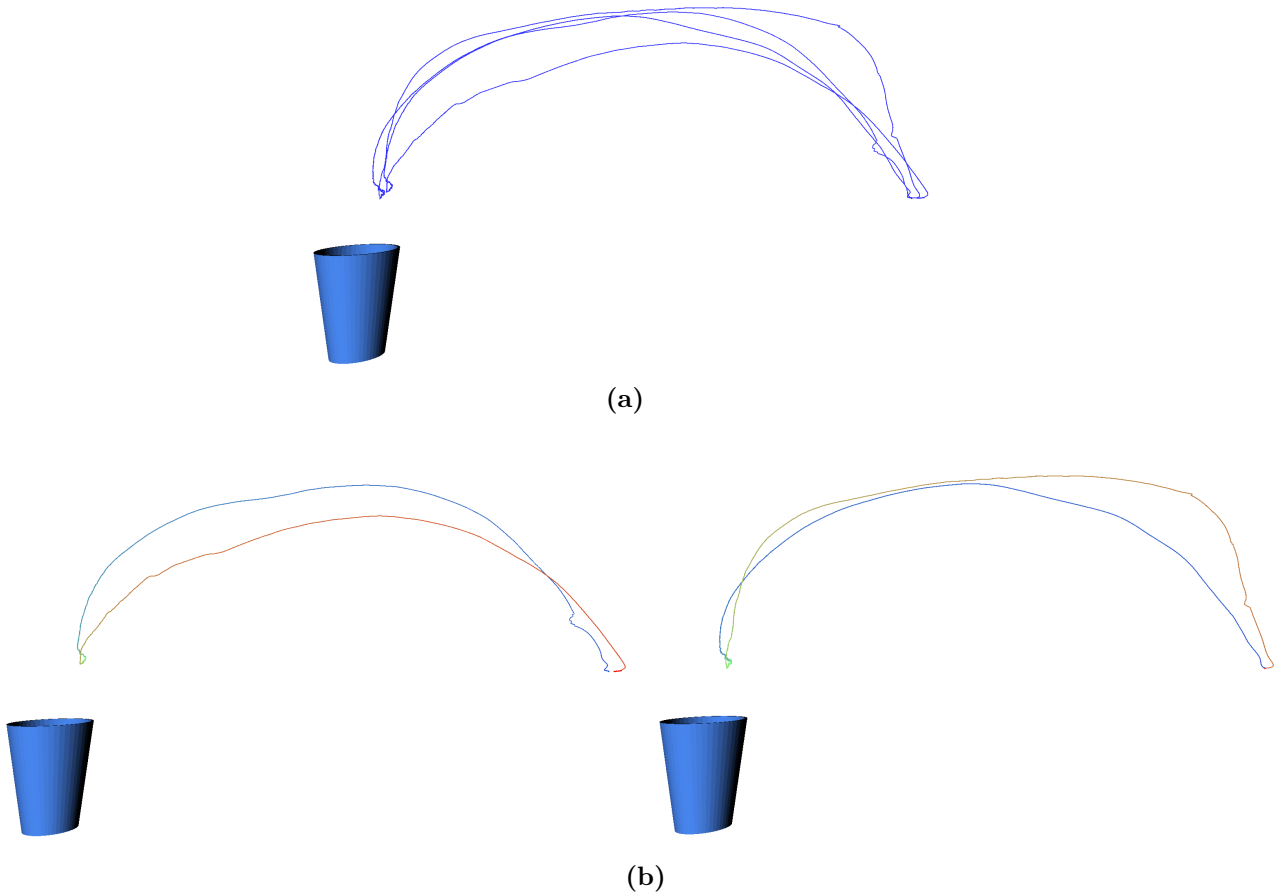
Figure 7.1: The visualization of extracted motions by the `extractor` with the parameter `show whole bag after extraction` set to `true`.

7.1.1 Extractor Verification

Each extraction is viewed by setting the `show whole bag after extraction` parameter to `false` for verifying the correctness of the automated extraction. Most false positives or false negatives can be immediately noticed. Different kinds of extraction failures are shown and explained in the following.

7.1.1.1 Two Motions Mistaken As One

Two pouring motions are falsely taken for a single one as seen in fig. 7.2a. In some cases adjusting the parameters does not correct this mistake. In such situations, when a solution for correcting false positives can not be found, at least a filter for sorting them out has to be implemented in order to keep the data clean when comparing motions. Therefore, a new constraint is added: If the bottle crosses the limit defined in `pouring y-axis toggle` more than twice on the way to or away from the glass, the sample counts as invalid. It means that the extraction based on the scale topic has failed in that motion.



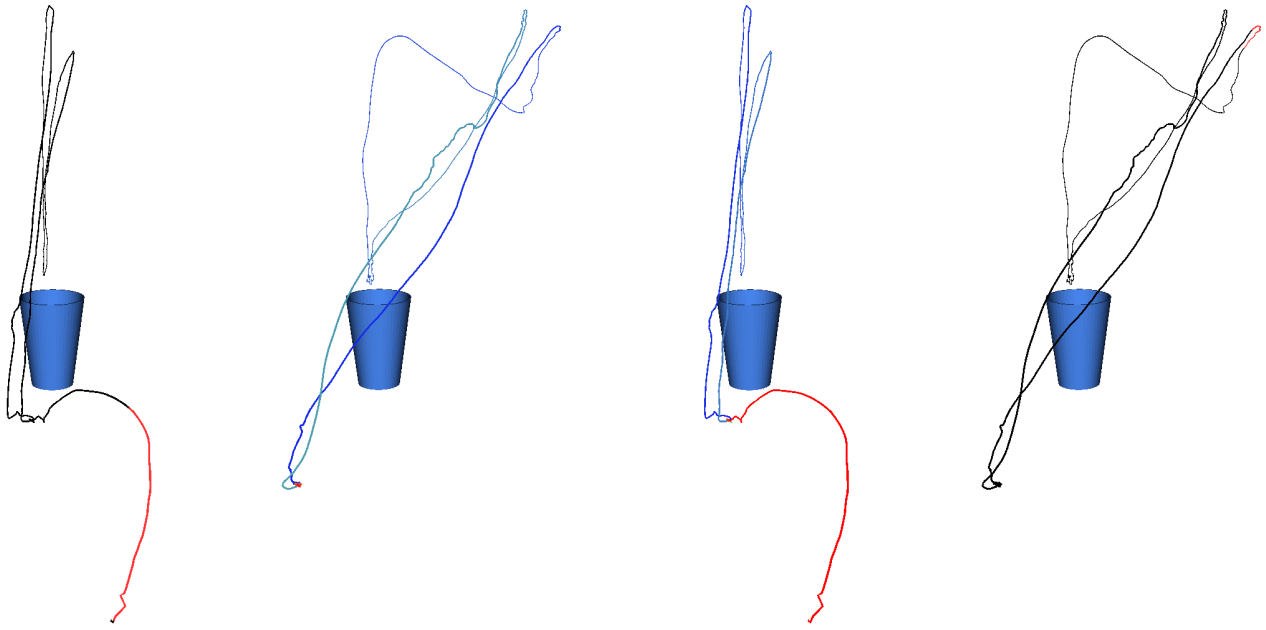
(a)

(b)

Figure 7.2: Extraction error fig. 7.2a corrected by changing the max weight difference parameter value from 0.001 kg to 0.01 kg fig. 7.2b.

7.1.1.2 Parameter Limits Overstepped

The motion counts as invalid, as soon as the bottle mouth is outside of the `pouring area x-axis limit` parameter. This leads to false negatives if the limit is set too close, and to false positives if the limit is set too high (fig. 7.3). The parameter is adjusted manually until the least amount of failed extraction is observed.



(a) False negative with a 0.15 m limit

(b) False positive with a 0.35 m limit

(c) Correct classification with a 0.20 m limit

Figure 7.3: Some extraction failures are corrected by changing the `pouring area x-axis limit` parameter. Black lines indicate an invalid motion with red parts being the waypoints crossing the limit. Red parts in valid motions (fig. 7.3c) indicate the waypoints at the beginning of the motion's duration.

Not all failed extractions can be explained by looking at their visualization. Some are analyzed further by testing their properties for outliers (e.g., amount poured below zero). Correctly extracted motions being marked as invalid are tolerated if no other solution is found because traversing an invalid motion with a real robot can be dangerous.

Four types of invalid motion extractions are defined as a result of the observations and prioritized in following order, to prevent multiple failure counts for the same extracted motion in the statistic.

1. Out of range (`pouring area x-axis limit`)
2. Not tilted enough (`minimal tilt angle`)
3. Negative amount poured
4. Motion repeated

Parameters (X, T, W, R)	Extracted	Valid	Failed	Reasons (X, T, P, R)
20, 50, 001, 2	216	187	29	17, 1, 9, 2
20, 50, 005, 2	220	195	25	17, 2, 6, 0
20, 50, 010, 2	220	196	24	17, 2, 5, 0
20, 50, 015, 2	220	196	24	17, 2, 5, 0
20, 50, 020, 2	222	197	25	17, 3, 5, 0
15, 75, 020, 2	222	194	28	22, 1, 5, 0
20, 75, 020, 2	222	197	25	17, 3, 5, 0
30, 75, 020, 2	222	203	19	11, 3, 5, 0
20, 50, 025, 2	222	197	25	17, 3, 5, 0
20, 50, 030, 2	222	197	25	17, 3, 5, 0
20, 75, 030, 4	186	154	32	16, 0, 9, 7
20, 50, 040, 2	222	197	25	17, 3, 5, 0
20, 50, 045, 2	222	198	24	17, 3, 4, 0
20, 75, 045, 2	222	198	24	17, 3, 4, 0
20, 50, 050, 2	222	198	24	17, 3, 4, 0
20, 75, 050, 3	220	193	27	17, 2, 7, 1
20, 50, 055, 2	222	198	24	17, 3, 4, 0
20, 50, 060, 2	222	198	24	17, 3, 4, 0
20, 50, 065, 2	222	198	24	17, 3, 4, 0
20, 75, 065, 4	192	159	33	16, 1, 10, 6
20, 50, 075, 2	220	196	24	17, 2, 5, 0
20, 50, 100, 2	213	183	30	17, 2, 8, 3

Table 7.1: Manual extraction parameter optimization results over all 13 recorded bags. The parameter values in the blue marked line produce the best quality output.
Parameters:
X-limit(cm), **T**ilt(deg), **W**eight diff.(g), **R**epeat count for stable weights (section 6.1.1).
Reasons for invalid motions:
Overstepped **x**-limit, not **t**ilted enough, negative amount **p**oured, **r**epeated motion

All parameters are adjusted and extracted motions manually checked repeatedly until a satisfying result is found. Changing the `maximal weight difference` parameter reduces the number of two pouring motions being extracted as one. This is observed after changing the value from 0.001 up to 0.01 kg in fig. 7.2. The final result consists of 198 correctly extracted and 24 invalid motions (table 7.1). One part of the invalid motions are truly failed pouring motions and the other part are wrongly extracted motions.

7.1.1.3 Exceptions

One unusual motion is seems wrong at first but can be used after filtering it (fig. 7.4). Such outliers show how fragile the recognition of human motions can as they are difficult to predict. More constraints can be set to filter out such unpredicted cases but they can also lead to less correct data. Which way is better depends on the goal of the data extraction. Since this motion is valid after the filtering no additional constraints are set here.

(a) The entire motion seems invalid at first sight

(b) Valid motion after filtering out all points farther than 35 cm away from the glass

Figure 7.4: Outliers are observed that seems to be a false positive at first but contain a valid pouring motion. Dealing with such unexpected motions is the most difficult part in automatic motion extraction.

7.2 Motion Analysis

In this section the recorded human motions are shown in different filter settings with multiple color mappings for spotting patterns and differences. The following conclusions are made based on the motion analysis:

1. The pouring point is always above and close to the outer edge of the glass.
2. The elevation of the bottle mouth is always lower on the way to the glass than on the return path, the difference decreases with slower movements.
3. Bottle spouts enable safer pouring from higher elevations.
4. When pouring from a high elevation, the start and end position are closer to the glass than midst pouring.

7.2.1 Regular

First the motions are filtered by leaving out all special configurations that include a bottle spout or purposely slow or high movement (fig. 7.5). Multiple patterns are immediately noticeable after filtering some outliers. The lowest points during pouring are close to the edge of the glass which can be explained as being a safety measure for two reasons:

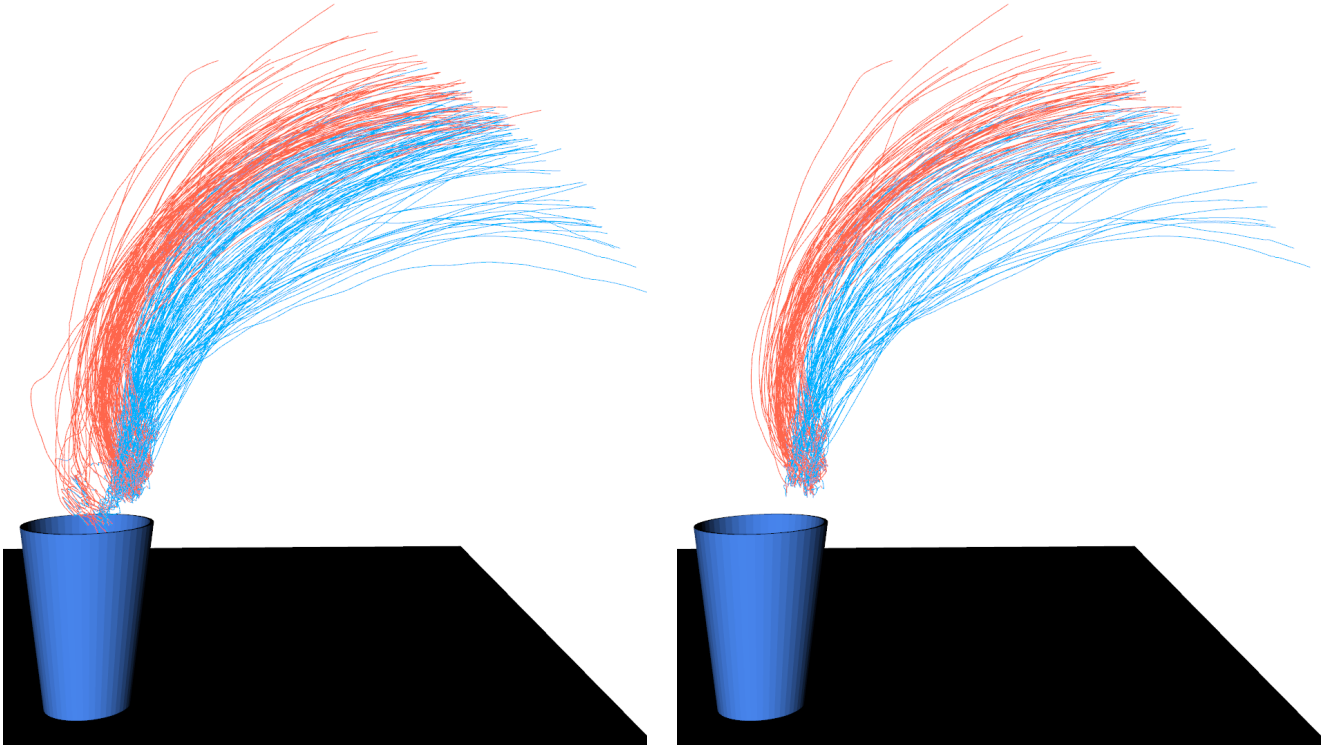
- The area for pouring without spilling is larger.
  - Closer towards the middle of the glass the chance of spilling forwards increases.
  - On the other hand it is impossible to spill when holding the bottle mouth inside of the glass but as liquid level in the glass rises it eventually touches the bottle which is not considered good as something in the glass can be transported to the bottle through the liquid (e.g., bacteria, other liquids). If the bottle is elevated above the glass to avoid contact with the

rising level of liquid the argument above holds true again.

- In case of vertical dripping the liquid still lands inside of the glass.

Using the direction color mapping another human movement pattern becomes visible: The elevation of the bottle mouth is always lower on the way to the glass than on the return path. This is due to the hand going up while tilting the bottle in order to keep the bottle mouth at the same level and then rotating the bottle back without going down with the hand to the initial elevation.

The reason for the outliers is usually the last pouring motion that empties the bottle which tends to break the usual pouring pattern. This is cause by less constraints in the motion because the pouring does not need to be stopped intentionally. It is instead often tilted more to shorten the pouring duration.



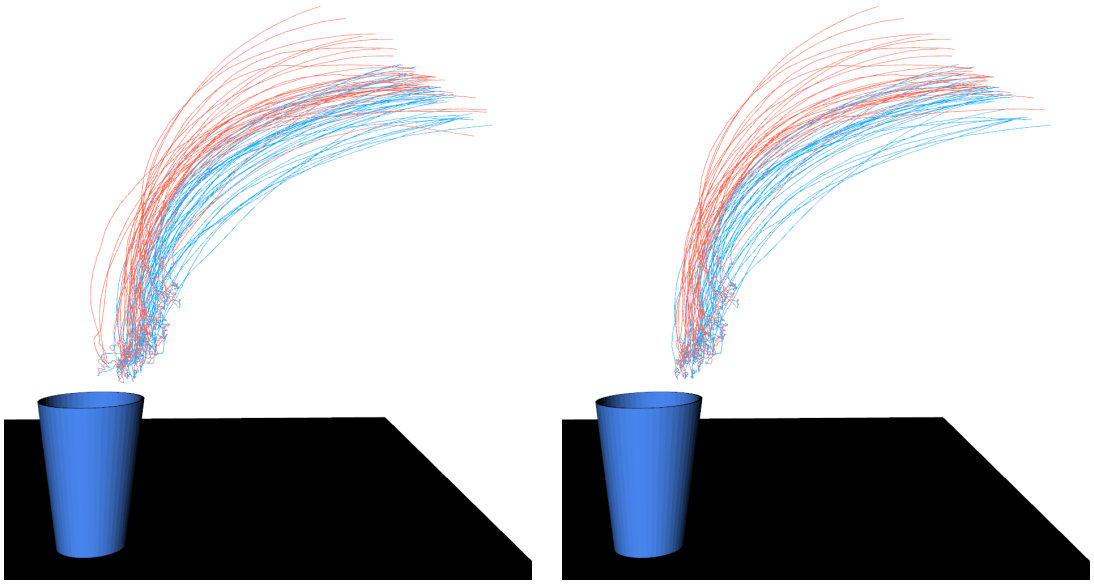
(a) 103 motions, no additional filters

(b) Outliers are mostly motions before bottle refills (amount poured < 300 ml, inital amount > 300) ml

Figure 7.5: Configuration: 1, 2 and 3, Filter: Max. 0.5 m from glass, Color: Movement towards (blue) and away (red) from bottle

7.2.2 Spout

Next the recorded motions using a bottle spout are analyzed (fig. 7.6). The pouring elevation is expected to be higher as the recorded points do not take the spout into consideration but it is not clearly observable. Most filtered motions are shifted to the right compared to the ones without the spout which is expected given the missing spout length. Another aspect that is observable for the first time, is that the pouring elevation differs between motions.

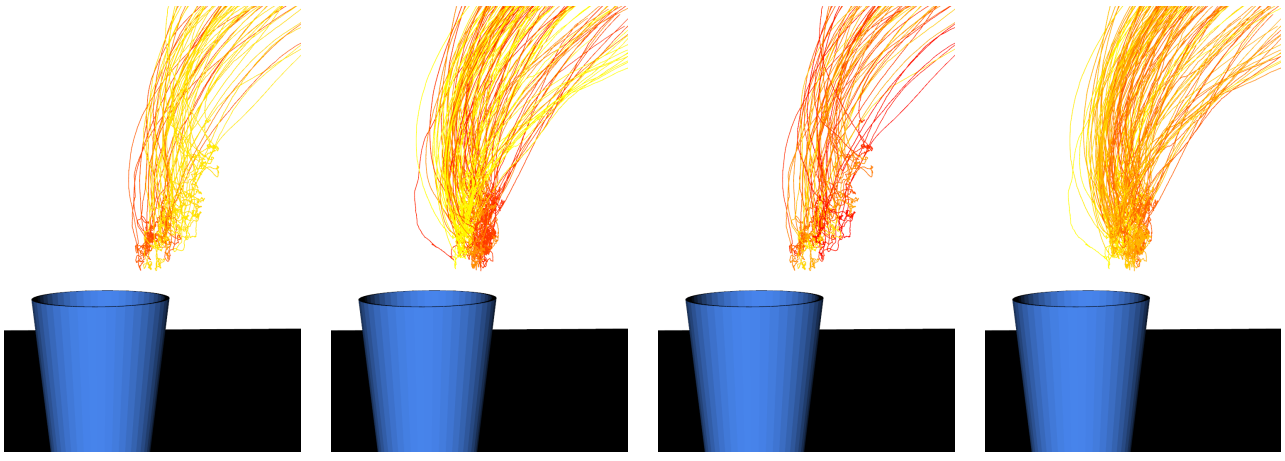


(a) 38 spout only motions

(b) 32 motions, outliers are filtered with the amount poured set to at least 60 ml

Figure 7.6: Extracted motions from configurations 7 and 8, filtered by distance (max. 0.5 m from glass).

The reason for different elevations in the spout motions is analyzed by mapping the color to different properties of the trajectory messages. Small indicators are seen when mapping it to the initial amount and the amount poured (fig. 7.7). The bottle elevation seems to increase with a decreasing pouring amount and an increasing initial amount but when applying the same color maps on the regular motions this pattern is not verified. The range of possible pouring elevations increases due to easier pouring with bottle spouts is the resulting explanation.



(a) 32 filtered trajectories with spout, color: amount poured

(b) 72 regular filtered trajectories, color: amount poured

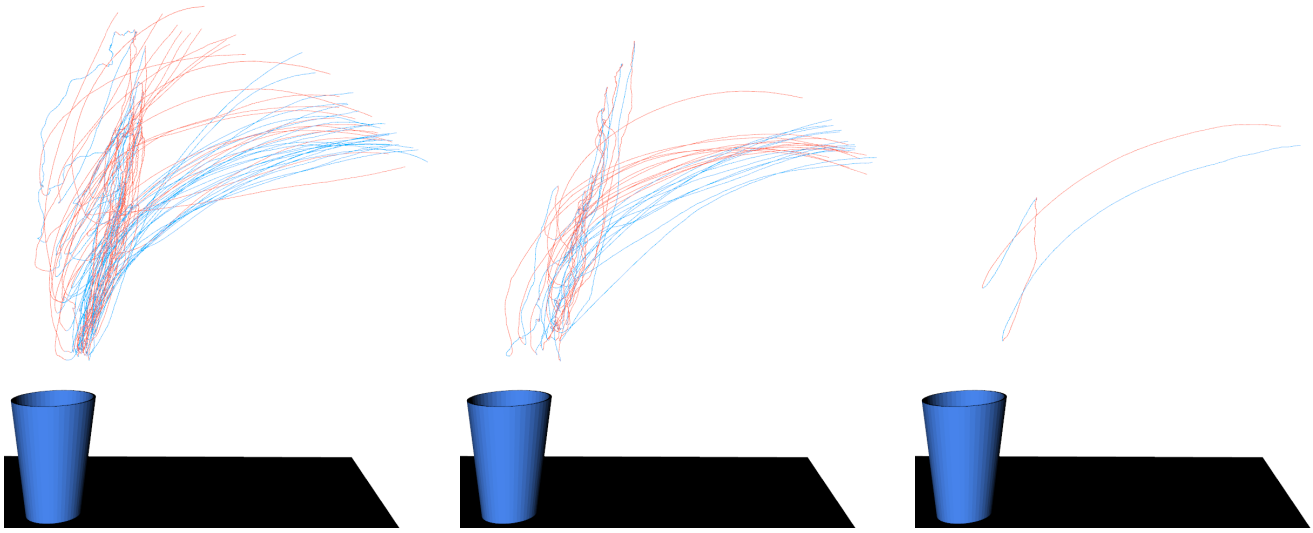
(c) 32 filtered trajectories with spout, color: initial amount

(d) 72 regular filtered trajectories color: initial amount

Figure 7.7: The color mappings by the initial amount and poured amount are compared. The color changes from yellow (smaller values) to red (higher values). The differences are clearer for the poured amount because in every configuration the same amount is poured approximately while the initial amount decreases steadily.

7.2.3 High

Another pattern is visible when looking at motions where the bottle is held as high as possible (fig. 7.8). Humans begin pouring at a rather close point before moving away once the liquid is flowing steadily to minimize the risk of spilling at the beginning of the motion. At the end of the motion the bottle is moved closer to the glass again in order to avoid spilling due to vertical liquid dripping after the break of the steady flow.



(a) All 24 motions, poured from as high as possible

(b) All 12 motions, poured from as high as possible with a spout

(c) Typical (no spout), high trajectory

Figure 7.8: Extracted high trajectories from configurations 6 and 9

7.2.4 Slow

Purposely slowly performed motions have a smaller difference between the forwards and backwards elevation than the motions performed at regular speed. This is explained by the human having more time to bring the hand back down to the starting position after pouring.

(a) All 21 valid motions, poured as slowly as possible

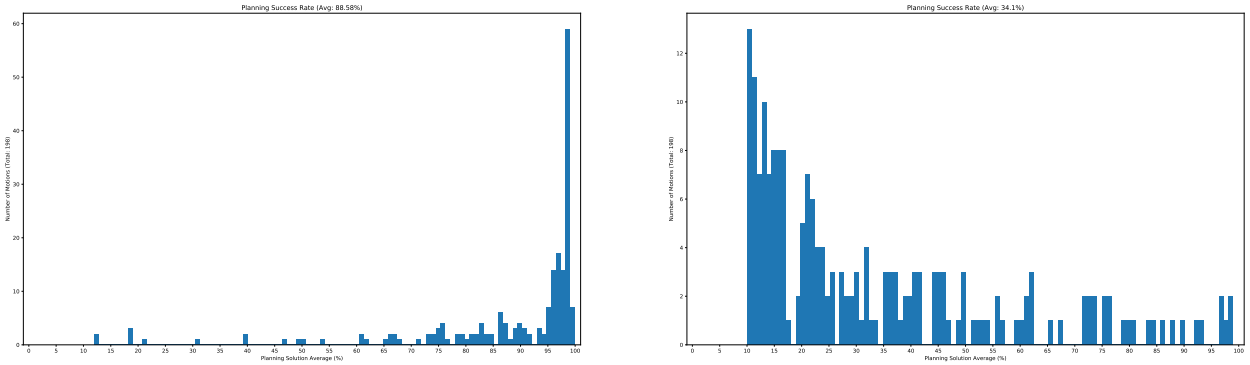
(b) Typical slow trajectory sample

Figure 7.9: Extracted slow trajectories from configuration 4 and 5

7.3 Imitation of Trajectories

Playing back the original motions on the robot is not easily done for multiple reasons. No motion plan is found in many cases which is why a method for analyzing different solutions is used to obtain an overview of the possibility of finding a motion plan. The method rotates the motion in steps of one degree around the glass and colors the points that are in the motion plan green, the others red, and motions where the starting point can not be reached are ignored and grayed out.

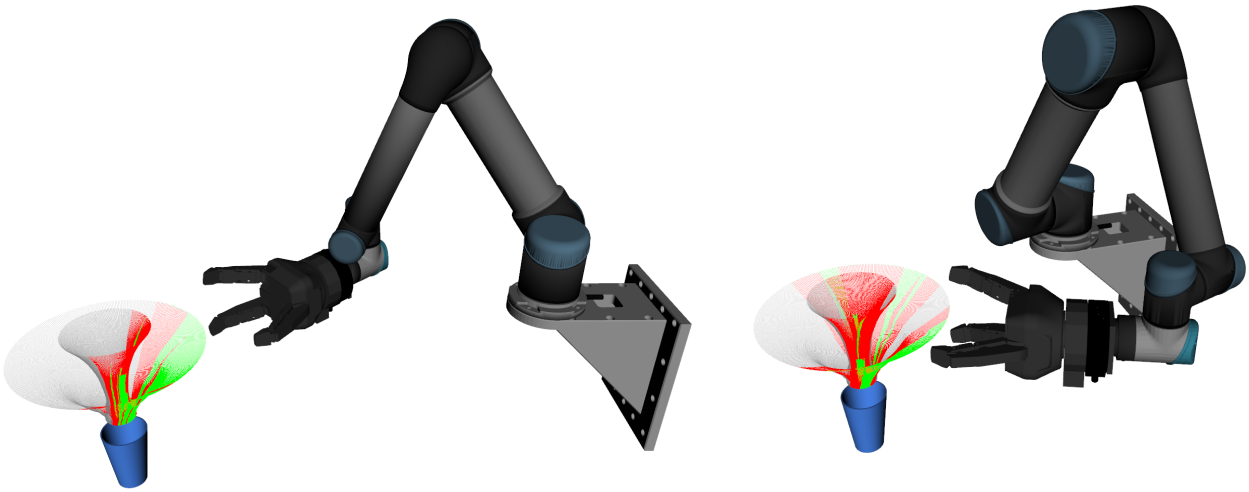
Figure 7.12 proves that solutions for smoother motions are significantly better and that the position of the glass relative to the robot causes changes in the solutions found as well. Smoothed motions sometimes lead to different results than the original motions (table 7.2) and the differences in smoothing outcomes vary with different amounts of kernels and for the same amount of kernels for different motions because sometimes overfitting occurs. The highest and the average computed solution (percentage) is stored for each motion from every rotation around the glass (in steps of 1°) in order to find suitable motions to play back on the robot after trying out a few motions unsuccessfully. An overview of this data is seen in fig. 7.10.



(a) Histogram for the best plans found for each motion

(b) Histogram for the average plan percentage, it shows that the high solutions in the plot on the left are not reached that often

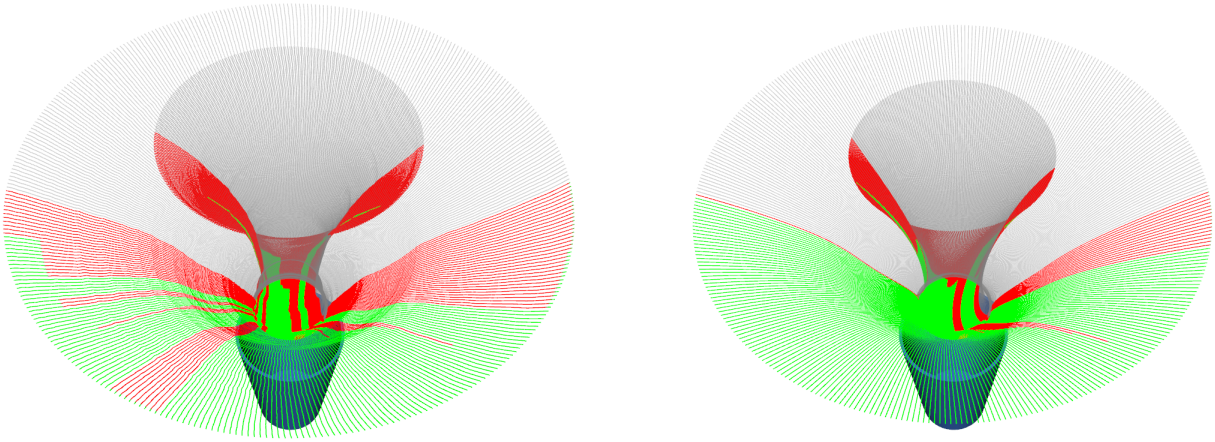
Figure 7.10: Motion plan percentages (rounded down) found for all recorded motions in 360 angles. Since the outcome of the plans depends on the position of the glass and the robotic arm these plots apply only to that specific environment the plans were computed in, and is not generalizable for motions. It is only made to give an overview of one case.



(a) Average: 42,45%, best: 99,89%

(b) Average: 19,99%, best: 99,78%

Figure 7.11: The planning solutions for a glass standing further away and closer to the robot are compared. The placement of the glass plays a big role in the possibilities of pouring.



(a) Average: 29,95%, best: 99,25%, original trajectory

(b) Average: 54,15%, best: 100%, smoothed trajectory with 10 kernels

Figure 7.12: A recorded trajectory is rotated and a solution for traversing it is computed for each rotation. Three colors differentiate between poses that can be traversed (green), poses that can not (red) and trajectories for which the robot can not find an IK solution for the starting point (gray).The first subtitle value is the percentage of waypoints that can be traversed on average out of all nongrey trajectories.

7.3.1 Trajectory Speed

When playing back a motion only using CCP the resulting movement of the robot is very slow and it trembles. After the joints' speed limits are set to their maximum it still can not keep up with the original motion duration. The input and output data is analyzed to find out what causes the slow movements. First the velocity is compared to the original velocity by using the `helper` for retrieving the gripper positions, transforming them to the bottle mouth and calculating the velocities based on the time steps given by CCP like in the equation in section 6.4.5. The resulting velocities (fig. 7.13) are clearly below the maximum joint velocity limits of around 3 m/s which is why the joint velocities and accelerations are evaluated next to receive clearer insights on the robot's slowness (fig. 7.14). The joint velocities are also far below the limit but the accelerations often reach their limit and change their direction quickly. This explains the slowness because with changing acceleration a robot can not gain high velocities. The changing accelerations are explained by the jitter in the original motions (fig. 6.6).

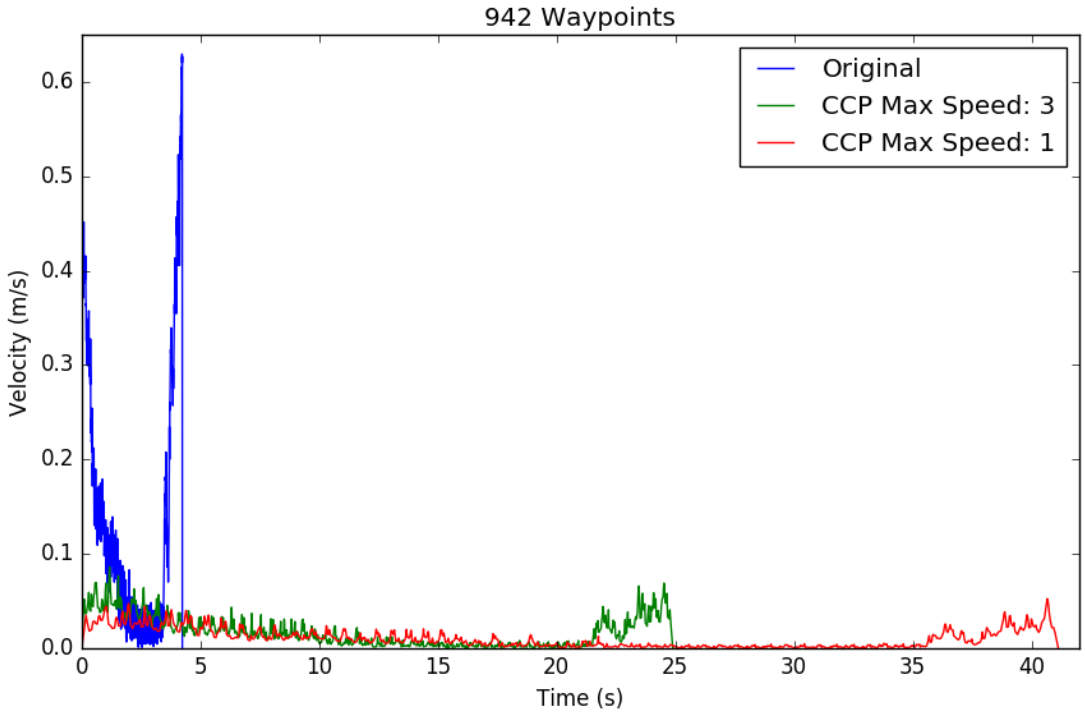


Figure 7.13: Every point of original trajectory is traversed, but the robot arm can not keep up with human speed using only CCP.

The waypoints of the original trajectory are filtered in hopes of smoothing the movement and speeding up the robot to traverse the motion in the original duration. CCP is executed and the number of points filtered out in between each point is adjusted automatically until the resulting duration matches the original as close as possible. The time of the last waypoint is compared to the time of its corresponding original point in order to correctly compare solutions below 100%. Figure 7.16 shows that the timing during of the motion is not in sync even if the overall durations match closely. The robot can not catch up to the original speed at first because it starts standing still while the original motion starts midst movement. In the last half of the duration the robot outruns the original motion which shortens the effective pouring time. The robot's motion during the last part lags behind when filtering out less points in order to match the points where the effective pouring is assumed. Increasing that filter (every 15th point) by one already makes the robot outruns the motion in the second half again. In order to solve this

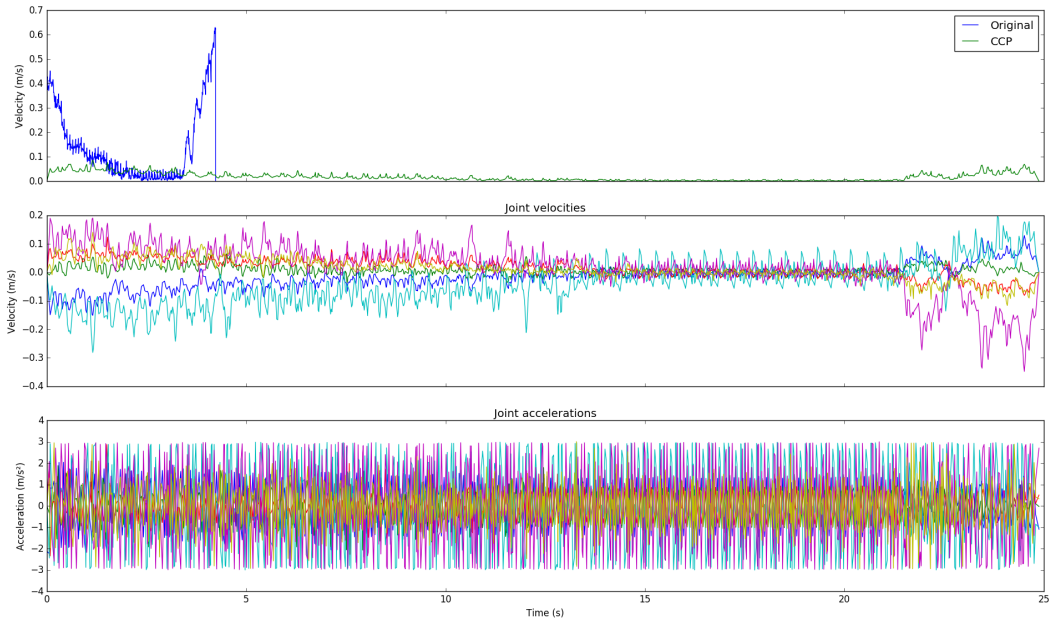


Figure 7.14: The velocities of the bottle are plotted on the top graph, below the velocities of each joint and on the bottom their accelerations for the same pouring motion. The reason for the slow movement is the quickly changing acceleration which again depends on the uneven original trajectory

problem the way of filtering points has to be changed to a more selective approach, e.g., filtering more points at faster parts and less at slower parts of the motion. Instead of following this approach further the Reflexxes library is used to focus on the velocities and assuring smooth movements which is not likely with the point filter approach alone.

7.3.2 Velocity adjustment

Given the joint trajectory values with their computed target velocities the first results of Reflexxes are unsatisfying as they do not follow the original motion points precise enough fig. 7.15. When changing the implementation to wait until each point is reached exactly, this problem is solved.

7.3.3 Real Pouring Test

Operations with liquids can be extremely dangerous for not waterproof robots. That is why the robot is manually moved to initial starting positions from which the planning success rate is high in order to be able to test traversing the recorded motions in reality. The glass is placed at a position that is easy to reach. The same motion is repeated with a closed bottle until a test with liquid is considered safe (photograph of setup seen on page 3).

The bottle is refilled after each test to ensure the same initial amount. No other motions can be tried because this runs they risk of breaking the orientation constraints of the gripper which happens infrequently or not finding suitable solutions for a long period after the robot moved to a different position.

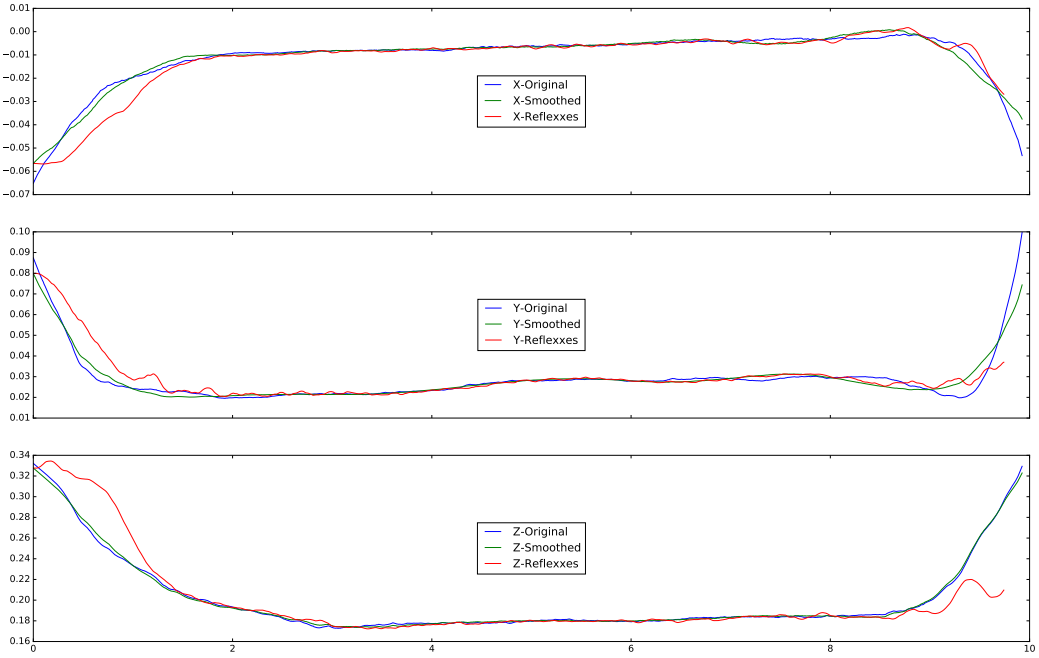


Figure 7.15: Reflexxes not catching up when given next waypoint directly after each cycle. After waiting for each point, the resulting trajectory can not be seen in the graph anymore because it lies directly on top of the smoothed one.

The accuracy of the scale is in the range of 2 g which can accumulate to an error of up to 8 g. The purpose of the test is to compare the impact of different parameter values on the outcome and how consistent it is.

While the skipping of every other point may seem promising during the experiment the robot could not find planning motions for it repeatably which make it worse for planning compared to the smoothed version.

Initial Amount	Amount poured (human / robot)	Kernels	Skipped points
640	56 / 10	20	1
640	56 / 33	0	1
640	56 / 116	0	0
640	56 / 44	20	0
640	56 / 98	20	0
640	56 / 42	20	0
640	56 / 64	20	0
640	56 / 58	20	0
640	56 / 64	20	0

Table 7.2: Tested Trajectories

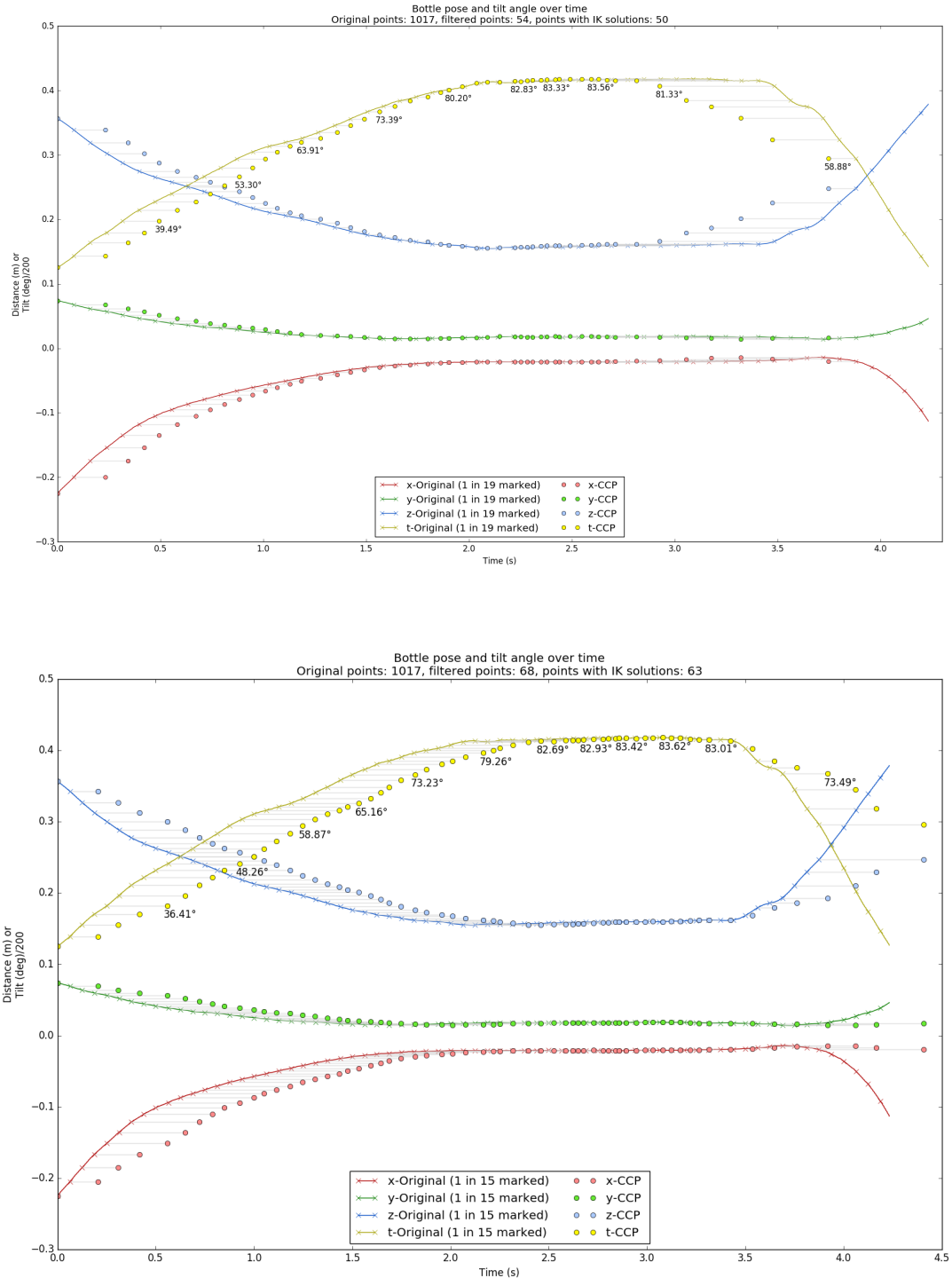


Figure 7.16: CCP output with same duration as original motion

8 Conclusion

This chapter presents the answers and results in section 8.1 corresponding the the questions and goals introduced in chapter 1. Section 8.2 describes encountered difficulties during the implementation phase in chapter 6. Future research that can be based on the concept (chapter 4) and prototype created in this thesis are given in the final section 8.3.

8.1 Answers and Result

The concept answering the main question in section 1.3 (how the task of consistent adaptive pouring can be solved) is presented in chapter 4. It states that changes in location and height of objects and the amounts of liquid are some of the most important properties of the environment needed to be known and to adapt to for successful pouring. Furthermore smooth and predictable motions are needed to gain acceptance and ensure safety.

To answer how they truly look like, real human pouring motions (as opposed to guided movements) are recorded (chapter 5), visualized (chapter 7) and finally played back on the robot which is not always possible, answering the last question of the thesis. It mainly depends on the location of the robot and the glass and the original velocity if a motion can be played back correctly. Moving along the exact waypoints of the original motion is usually possible from at least one angle but because the original speed can barely be matched motions should be smoothed first.

8.2 Encountered Difficulties

Research in larger areas like robotics where multiple components have to work together leads to a variety of problems that do not have much to do with the task at hand directly. In this section some of these problems are described that should be known when working on similar projects.

8.2.1 Software Issues

Open source software enables the creation of large frameworks with the help of motivated programmers but tends to lack good documentation as few have the motivation to create it. A few of these problems are encountered during the implementation described in the following.

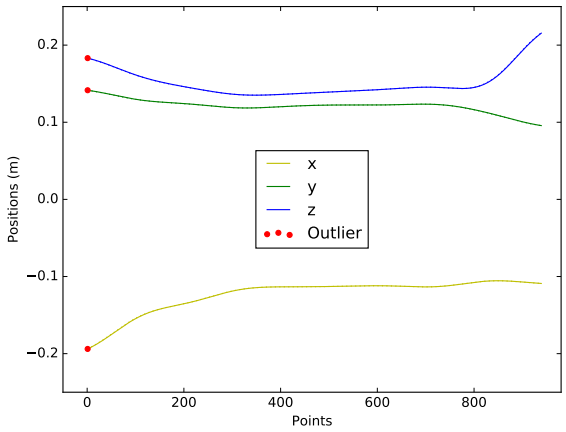
8.2.1.1 Documentation

There are multiple conversion methods in the `tf` namespace but it is found to be inconsistent as not all conversion can be made easily in both direction. This may have its reasons but is not intuitive for beginners and it is not easy to find the missing conversions somewhere else. One solution is found by creating a `tf::Matrix3x3` object using a quaternion and then calling its `.getRPY` method.

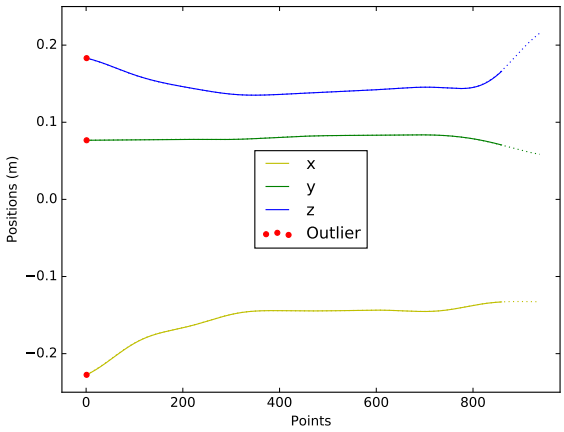
Another problem is the documentation of position constraints in MoveIt! because they have many parameters without any examples. How to visualize the set constraints is not explained or why they are unexpectedly removed after setting new targets for the robot. The target is set before the constraints to avoid this problem but even then undocumented exceptions happen.

8.2.1.2 Point duplicate

In some cases the path computed by CCP contained more points than the recorded path. This causes a problem when applying the original timestamps to the points because there are more points than timestamps. CCP generates additional points for a trajectory each time the next point is farther away than a given minimal distance, but this is not the case with a minimal distance of 10 cm using the original points. Therefore a python script is written that compares the generated trajectory to the original one and marks the first outlier to get an idea what the reason for it could be by looking at the area in which the outlier occurs (fig. 8.1). This leads to the finding that the first point is duplicated. Other trajectories are examined to see if this is a recurring issue and that is confirmed (8.1b). This fact goes unnoticed for long because CCP never finds a 100% solution for the tested trajectories up until they are smoothed.
To prevent this, the first original trajectory point is removed in the code assuming that CCP takes the starting position of the arm into account. That assumption works because when removing the starting point, it appears in the output again.



(a) Plot of computed trajectory (gripper position) with one point more than the original.



(b) The first point is also computed twice in cases where CCP does not find solutions to traverse all points.

Figure 8.1: After an error due to CCP returning more points than the original trajectory had, the input and output is plotted and outliers marked. This leads to the realization that the current point of the gripper is automatically added.

8.2.1.3 Reflexxes Steers into Wrong Direction

At first, after each output of the used method from the Reflexxes library, the next point of the given trajectory is set as the new target without waiting to reach the previous one. If the previous target is not reached it is assumed that the output is the closest point possible to it. This way a smoother gripper motion is expected. It is also expected that the robot can not match the speed in the beginning because it starts from a standing position while the bottle in the original motion is already midst movement at the first joint trajectory point but assumed not to influence the pouring outcome too heavily.

But the results are extremely shaky motions and unexpected behavior (as seen before in fig. 7.15). The joints are sometimes turned into the opposite direction and the velocities do not match the targets.

This lead to a long period of code reviewing because an issue in the implementation was suspected to cause the unwanted behavior. But it turns out Reflexxes' method tries to compensate the slowness by moving the arm back and then forth to reach the desired speed. But if the target goal changes during those movements this results in an even bigger offset. It is solved by waiting until the target is reached before continuing with the next point. This leads to more poses overall, but all poses can be reached precisely and the velocities are still matched closely because the robot has more time to accelerate in the beginning.

The resulting accelerations from Reflexxes are either zero or the maximal value which is another unexpected and not mentioned in the documentation. A reason for this could not be found.

8.2.1.4 Added Frame Excluded

CCP only uses the frames in the URDF, not additionally inserted ones like the glass or the bottle in one case. Therefore the pose in the glass frame has to be transformed to another frame in the URDF. The same problem is encountered in the Rviz, when the bottle does not move because it is not known by the MoveIt! plugin.

8.2.1.5 Exchanging Data between Python3 and Python2

Calling a method in the Python3 smoothing script and passing a number to it is easily implemented, but the passing and receiving of trajectory messages is a bigger challenge than expected and a workaround is implemented by writing, changing and reading the data using a CSV file. The exact process is modeled in fig. 8.2.

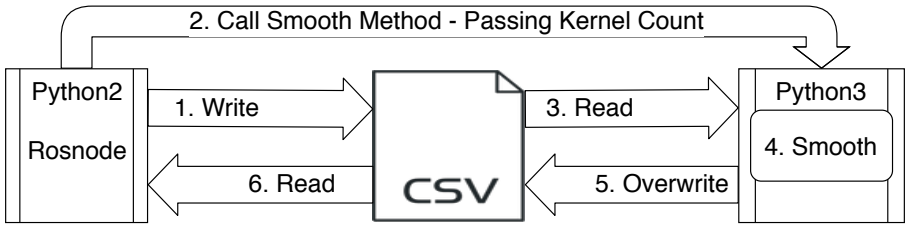


Figure 8.2: The workaround for Python2 node calling a Python3 method while exchanging trajectory data through a CSV file.

8.2.1.6 Error Message

After some refactoring the compiler returned following error:

```
terminate called after throwing an instance of 'std::runtime_error'
  what():  Duration is out of dual 32-bit range
```

It turns out a node was renamed, but its type value in a `launch` file still carried its old name which led to the meaningless error.

8.2.2 Unexpected Robot Behavior

Working with the real robot adds components to the implementation process that do not need to be addressed while only executing its movements in simulation. The robotic arm and the gripper have separate drivers that can be launched with different parameter values.

8.2.2.1 Package-Specific Parameters

The gripper is able to perform different types of grasping. In the case of household bottles the gripper mode should to be set to `basic`. Setting the mode to the wrong value, e.g. `pinch`, makes it impossible to use predefined grasps of another mode.

8.2.2.2 Sudden Movement at Beginning

Right before starting the pouring motion the robot makes a sudden motion. This is due to the approximate goal planning which is set in order to move to a pose not perfectly exact to the goal pose when not finding the exact plan.

8.3 Outlook

Ideas for future work based on the components implemented in the prototype are presented in this section. This spans from machine learning, analyzing other motions to suggestions for new features in the implementation.

8.3.1 Learning

The following two steps are suggested for using machine learning approaches to create generic pouring solutions based on the results of this work.

- **Recording more human motions**
  A strength of machine learning algorithms lies in the processing of large data collections, therefore a lot more samples have to be recorded before satisfiable results are expected. This approach can benefit from the implemented `analyzer` and the described recording setup in chapter 5.
- **Pouring with a real robot using self-optimization**
  An initial rudimentary motion set with multiple adjustable parameters can be created using the recorded data in this thesis. The robot can then repeat the pouring motions given a goal and optimize the parameters. Reaching a specified weight of a glass after pouring is an example of such a goal. The weight can be automatically measured by a connected USB-Scale and the robot could then empty the glass on its own in order to fully automate this process. The behavior not being transferable to other robots may be a downside of this approach and the selection of the adjustable parameters is also non-trivial.

8.3.2 Motions

Lots of other tasks can be recorded and executed with the robot besides pouring. For automated motion extraction the `analyzer` class will most likely have to be rewritten, but given complete motion samples the created WUI is a simple interface for quickly analyzing and even traversing different motions using the `analyzer` and `pourer` classes. Extending the `helper` class to more filtering methods enables analyzing additional properties. A couple of different motions to try these classes out with are suggested below.

- Throwing
- Drawing

- Gesturing
- Stirring

8.3.3 Implementation

Following improvements and additions for the implemented prototype are suggested given more time.

The possibilities of merging existing methods should be analyzed to make the code more compact, reusable and extensible.

8.3.3.1 Automated Extraction

New parameters, better parameter values and new extraction rules can be implemented after analyzing more recorded motions further. Also an optimizer could be added which computes the best parameter values given a set of correctly labeled samples for more automation.

8.3.3.2 Pouring Service

A method should be implemented for stopping the pouring process when it is taking too long to improving the efficiency of testing different motions.

The constraint values of the gripper orientation are static but can be set dynamically depending on the initial amount in the bottle. The less liquid is inside the bottle, the more freedom the gripper can have because it is less likely to spill.

Bibliography

[1] "Executive Summary World Robotics 2017 Industrial Robots," International Federation of Robotics, visited on 17.07.2018. [Online]. Available: https://ifr.org/downloads/press/Executive_Summary_WR_2017_Industrial_Robots.pdf

[2] Ito, Atsushi and Noda, Yoshiyuki and Tasaki, Ryosuke and Terashima, Kazuhiko, "Outflow Liquid Falling Position Control Considering Lower Pouring Mouth Position with Collision Avoidance for Tilting-Type Automatic Pouring Machine," *Materials Transactions*, vol. 58, no. 3, pp. 485–493, 2017.

[3] "Liquid handling robot," Wikipedia, visited on 17.07.2018. [Online]. Available: https://en.wikipedia.org/wiki/Liquid_handling_robot

[4] "Universal Robot UR5," Universal Robots, visited on 10.07.2018. [Online]. Available: https://www.universal-robots.com/products/ur5-robot/

[5] "Robotiq Force Torque Sensors," Innovative Total Solutions, visited on 10.07.2018. [Online]. Available: https://itsl.ie/wp-content/uploads/2016/10/Specsheet-FT150-English.pdf

[6] M. Bestmann and F. Wasserfall and N. Hendrich and J. Zhang, "Replacing cables on robotic arms by using serial via Bluetooth," in *2017 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, Dec 2017, pp. 189–195.

[7] "3-Finger Adaptive Robot Gripper," Robotiq, visited on 10.07.2018. [Online]. Available: https://robotiq.com/products/3-finger-adaptive-robot-gripper

[8] "Kinect," Wikipedia, visited on 17.07.2018. [Online]. Available: https://en.wikipedia.org/wiki/Kinect

[9] Tolani, Deepak and Goswami, Ambarish and Badler, Norman I., "Real-time Inverse Kinematics Techniques for Anthropomorphic Limbs," *Graph. Models Image Process.*, vol. 62, no. 5, pp. 353–388, Sep. 2000. [Online]. Available: http://dx.doi.org/10.1006/gmod.2000.0528

[10] Quigley, Morgan and Faust, Josh and Foote, Tully and Leibs, Jeremy, "ROS: an open-source Robot Operating System."

[11] "Packages," Open Source Robotic Foundation, visited on 15.05.2018. [Online]. Available: http://wiki.ros.org/Packages

[12] "msg," Open Source Robotic Foundation, visited on 15.05.2018. [Online]. Available: http://wiki.ros.org/msg

[13] D. Erickson, "Standards for Representation in Autonomous Intelligent Systems," p. 22, 2005.

[14] Mueller, Rainer and Vette, Matthias and Kanso, Ali, "Comparison of practically applicable mathematical descriptions of orientation and rotation in the three-dimensional Euclidean space," in *Tagungsband des 3. Kongresses Montage Handhabung Industrieroboter*, Schueppstuhl, Thorsten and Tracht, Kirsten and Franke, Joerg, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2018, pp. 123–130.

[15] "Conversion between quaternions and Euler angles," Wikipedia, visited on 13.07.2018. [Online]. Available: https://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles

[16] "transform_datatypes.h," Open Source Robotic Foundation, visited on 13.07.2018. [Online]. Available: http://docs.ros.org/kinetic/api/tf/html/c++/transform__datatypes_8h.html

[17] "rosbridge_suite," Open Source Robotic Foundation, visited on 13.07.2018. [Online]. Available: http://wiki.ros.org/rosbridge_suite

[18] "The Standard ROS JavaScript Library," Open Source Robotic Foundation, visited on 13.07.2018. [Online]. Available: http://wiki.ros.org/roslibjs

[19] "Manual and Documentation (Type II, Version 1.2.6)," Reflexxes Motion Libraries, visited on 07.05.2018. [Online]. Available: http://www.reflexxes.ws/software/typeiirml/v1.2.6/docs/index.html

[20] John Wang and Edwin Olson, "AprilTag 2: Efficient and robust fiducial detection," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, October 2016, pp. 4193–4198.

[21] Z. Pan and D. Manocha, "Motion planning for fluid manipulation using simplified dynamics," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct 2016, pp. 4224–4231.

[22] L. Rozo and P. Jiménez and C. Torras, "Force-based robot learning of pouring skills using parametric hidden Markov models," in *9th International Workshop on Robot Motion and Control*, July 2013, pp. 227–232.

[23] Z. Pan and D. Manocha, "Feedback Motion Planning for Liquid Transfer using Supervised Learning," *CoRR*, vol. abs/1609.03433, 2016. [Online]. Available: http://arxiv.org/abs/1609.03433

[24] M. Kennedy and K. Queen and D. Thakur and K. Daniilidis and V. Kumar, "Precise dispensing of liquids using visual feedback," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sept 2017, pp. 1260–1266.

[25] "KUKA Robot pours a beer at Hannover Messe 2017," YouTube, visited on 15.07.2018. [Online]. Available: https://youtu.be/c6mqv4vf2mg

[26] Constantin, Peter and Foias, Ciprian, *Navier-stokes equations*. University of Chicago Press, 1988.

[27] Z. Pan and C. Park and D. Manocha, "Robot motion planning for pouring liquids," *Proceedings International Conference on Automated Planning and Scheduling, ICAPS*, vol. 2016-January, pp. 518–526, 2016.

[28] Yongqiang Huang and Yu Sun, "Learning to Pour," *CoRR*, vol. abs/1705.09021, 2017. [Online]. Available: http://arxiv.org/abs/1705.09021

[29] C. Do and W. Burgard, "Accurate Pouring with an Autonomous Robot Using an RGB-D Camera," *Proceedings of the 15th International Conference on Intelligent Autonomous Systems*, June 2018.

[30] Pierre Sermanet and Corey Lynch and Jasmine Hsu and Sergey Levine, "Time-Contrastive Networks: Self-Supervised Learning from Multi-View Observation," *CoRR*, vol. abs/1704.06888, 2017. [Online]. Available: http://arxiv.org/abs/1704.06888

[31] I. Yanokura and M. Murooka and S. Nozawa and K. Okada and M. Inaba, "Variance Based Trajectory Segmentation in Object-centric Coordinates," *Proceedings of the 15th International Conference on Intelligent Autonomous Systems*, June 2018.

[32] Tatiana Lopez-Guevara and Nicholas K Taylor and Michael U Gutmann and Subramanian Ramamoorthy and Kartic Subr, "Adaptable Pouring: Teaching Robots Not to Spill using Fast but Approximate Fluid Simulation," in *Proceedings of the 1st Annual Conference on Robot Learning*, ser. Proceedings of Machine Learning Research, Sergey Levine and Vincent Vanhoucke and Ken Goldberg, Ed., vol. 78. PMLR, 13–15 Nov 2017, pp. 77–86. [Online]. Available: http://proceedings.mlr.press/v78/lopez-guevara17a.html

[33] "PhaseSpace Impulse X2E," PhaseSpace, visited on 10.07.2018. [Online]. Available: http://phasespace.com/x2e-motion-capture/

[34] Mcsheery Tracy D and Black John R and Nollet Scott R and Johnson Jack L and Jivan Vinay C, "Distributed-processing Motion Tracking System For Tracking Individually Modulated Light Points," United States Granted Patent US 6 324 296 B1, 2001. [Online]. Available: https://lens.org/037-222-783-825-603

[35] "How to define ROS kinetic to use python3 instead of python2.7?" ROS Answers, visited on 14.07.2018. [Online]. Available: https://answers.ros.org/question/237613/how-to-define-ros-kinetic-to-use-python3-instead-of-python27/

[36] "Bootstrap," Bootstrap, visited on 14.07.2018. [Online]. Available: https://getbootstrap.com/

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Masterarbeit im Studiengang Master Wirtschaftsinformatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und, dass die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Hamburg, den _____ Unterschrift: _____