WROCŁAW UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF ELECTRONICS, PHOTONICS AND MICROSYSTEMS

FIELD OF STUDY: Control Engineering and Robotics

BACHELOR THESIS

TITLE OF THESIS:

ROBOTIZATION OF THE CAPPUCCINO PREPARATION PROCESS WITH COMPUTER VISION FEEDBACK

AUTHOR:

EMILIA SZYMAŃSKA

SUPERVISOR:

JANUSZ JAKUBIAK, PHD

CONSULTANT:

PROF. JOSIE HUGHES

WROCŁAW 2022

To the CREATE lab team and all E3 students, who supported me on my coffee adventure and bravely stood up to the challenge of drinking all the experimental cappuccinos.

Contents

1 Introduction **3**
- 1.1 Subject overview and analysis . . . 4
- 1.2 Thesis outcome . . . 5
- 1.3 System design . . . 5

2 Hardware **7**
- 2.1 General overview . . . 7
- 2.2 Equipment . . . 8

3 Software **15**
- 3.1 General overview . . . 15
- 3.2 Robot Operating System pipeline . . . 15
- 3.3 Computer vision . . . 32
- 3.4 Supplementary software elements . . . 39

4 Optimization **41**
- 4.1 General overview . . . 41
- 4.2 Optimization methods . . . 42
- 4.3 Results . . . 44

5 Conclusions **49**

References **51**

A Technical drawings of custom 3D elements **55**

Chapter 1

Introduction

Food engineering is a science sector that offers a variety of opportunities for possible improvements in terms of technology integration. Research carried out in this area requires high repeatability and accuracy in experiments whose main objective is to understand both the physical and chemical reactions occurring in food or drink preparation [Jur06, TM09]. Due to the challenges resulting from high stochasticity, automation is necessary to be applied to as many aspects as possible to reduce the likelihood of external factors affecting the preparation process. Robotics provides one plausible solution for automating food production and scientific experiments, allowing for precise repetition as well as intelligent data capture with further analysis [KKI18, IKK17]. Investigating the foam creation of powdered hot beverages is one of the tasks in which robots can assist. This area is of high importance for the drinks industry and consumers [SSM+20, Sch07], with many studies focusing on the understanding of the foams production and formation. High complexity and multi-layered dependencies of this process make the resulting foam hard to assess. Robots analysing and optimizing food have used a variety of evaluation solutions – user feedback [JHTI20], salinity sensors [SHIH21] or tactile assessment [SIH21, SMCC+19] – yet there has not been a significant exploration of the use of computer vision as a means of providing rapid feedback to the food optimization process.

An idea to create a *Robot Food Scientist* with an ability to automatically prepare food and drinks with various input parameters, evaluate the output and optimize its creation emerged. A custom pipeline would be developed to decide on the product quality, using computer vision to automatically assess predefined properties. Introducing closed-loop control with local optimization of the preparation and response to undesired food characteristics would be the last – but not least – stage of the *Robot Food Scientist* project. However, a case study with exploration of the black-box optimization techniques giving an insight into identifying the optimal parameters would need to be examined beforehand. Cappuccino, being an inexpensive and not highly complex drink, seemed to be an appropriate starting point.

1.1 Subject overview and analysis

Cappuccino, a coffee drink originating in Italy, is nowadays served in cafes all around the world, gaining on popularity since early 1900s. According to the traditional recipe [Cib], preparation process starts with brewing coffee in a moka pot. Around thirty milliliters of the final product of this stage (called espresso) is mixed with 60 ml steamed milk and topped with milk foam, forming a 1-2 cm layer in a 150 ml cup. This foam, namely microfoam [con20], has quite specific requirements – it should be shiny, slightly thickened, consisting of minuscule and evenly distributed bubbles.

As the proper preparation demands some dedication, powdered substitute [Hof00] is a tempting alternative for people with time shortage or those who do not have necessary equipment and skill. Obtaining the coffee drink is then as simple as pouring hot water into the cup with powder while stirring. Mixing needs to be continued for some time to get rid of the powder clumps and to form a satisfactory layer of milk foam. Because of many simplifications in this case, preparation of powdered version of cappuccino has been selected as a starting point for exploration in the food sector optimization, with a prospect for future developments.

Although the aforementioned preparation process is quite straightforward for a human, implementing its steps with a feedback control loop for an automata turns out to be a multilayered challenge. Imitating the humanlike stirring motion or the real time powder clump squishing are just two examples of complicated machine behaviours which are just a small piece of a wider topic of robotic cooking. There has been some research exploring robots' utility in a kitchen [BKK+11, BBR11, SRLS16], however this area still requires further investigation due to the subject complexity. Senses of taste and smell for machines are not developed to such an extent as sight and hearing which can already be satisfied with cameras and microphones, with ongoing exploration towards tactile sensors. Cameras are widely available, relatively cheap data-gathering devices, therefore the evaluation of the prepared drinks or meals mainly relies on vision feedback.

It is quite challenging to make the robot adjust its actions while cooking without longer delays. Balance in sensors positioning in such a way that they do not interfere with the robot in carrying out its task, and that at the same time they collect all needed data is not easy to maintain. It may be necessary for the robot to move away or to stop in the middle of an action to take proper photos. What is more, image processing might take a while to compute the output, especially if the image quality and resolution are high. Feeding an artificial neural network without utilizing a dedicated processor – either Graphic Processing Units or Neural Network Accelerators – may be a reason of the delay in decision making [LC20].

1.2 Thesis outcome

Taking all the above into consideration, a robotic system for cappuccino preparation was developed. The project was conducted at Computational Robot Design and Fabrication Lab (CREATE Lab) at École Polytechnique Fédérale de Lausanne during an EPFL Excellence in Engineering research program. The laboratory provided necessary equipment i.a. a robotic arm, microcontrollers, cameras, servos, motors and 3D printers. Because of the simplification reasons and the resources offered by Nestlé S.A., the study aimed at carrying out experiments with powdered cappuccino. This thesis outlines the implementation of these stages with division into three main sections: descriptions of hardware, software and optimization process. After preparing a coffee, the system evaluates it and takes proper actions to improve it if needed. Optimization in search for input parameters resulting in the best cappuccino was performed as the final step of the study. Overall, the project consisted of the following stages:

- designing, 3D-printing and arranging the hardware setup,
- programming and testing the work of the robotic arm and other programmable hardware elements,
- proposing a computer vision approach towards coffee quality assessment,
- adding a closed-loop control,
- creating a software pipeline combining all steps needed to prepare cappuccino,
- carrying out multiple experiments in accordance with various optimization methods.

1.3 System design

Two equally important aspects – software and hardware – needed to be considered when designing the system. General steps necessary for powdered cappuccino preparation (Fig. 1.1) led to a designation of the devices included in the project, schematically shown in Fig. 1.2. Water and powder dispenser were added and automated to allow for efficient collection of the ingredients. Robotic arm moved the cup between devices, with its multifunctional end effector handling a stirrer and a camera. Another camera, taking photos from the side perspective, was placed vis-à-vis water dispenser so that it can monitor the cup content.

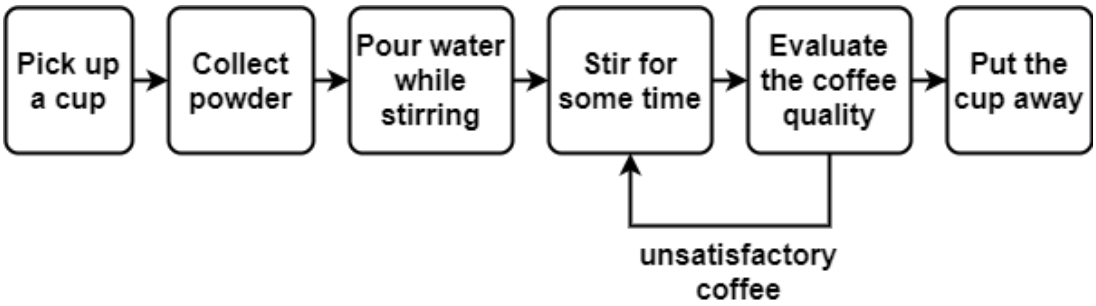


Figure 1.1 General coffee preparation steps

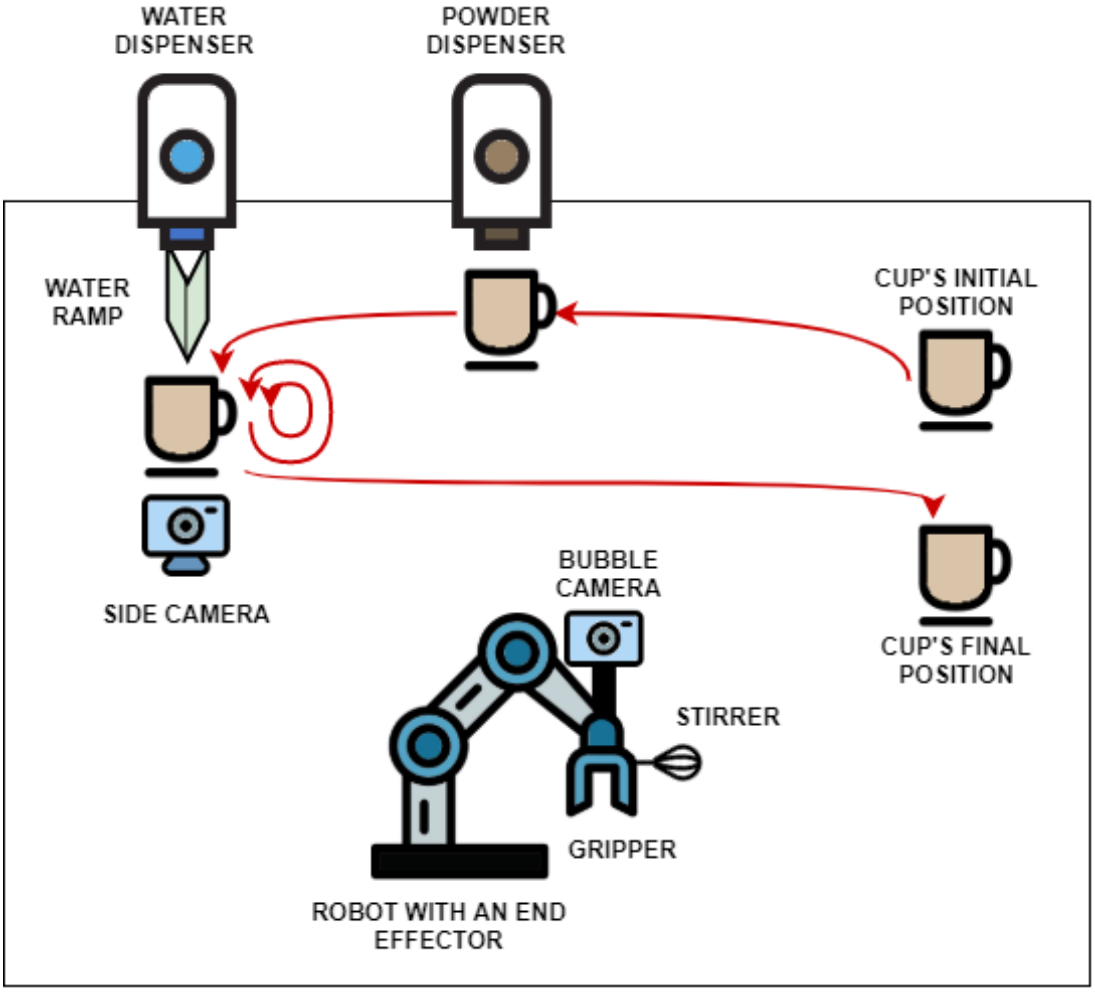


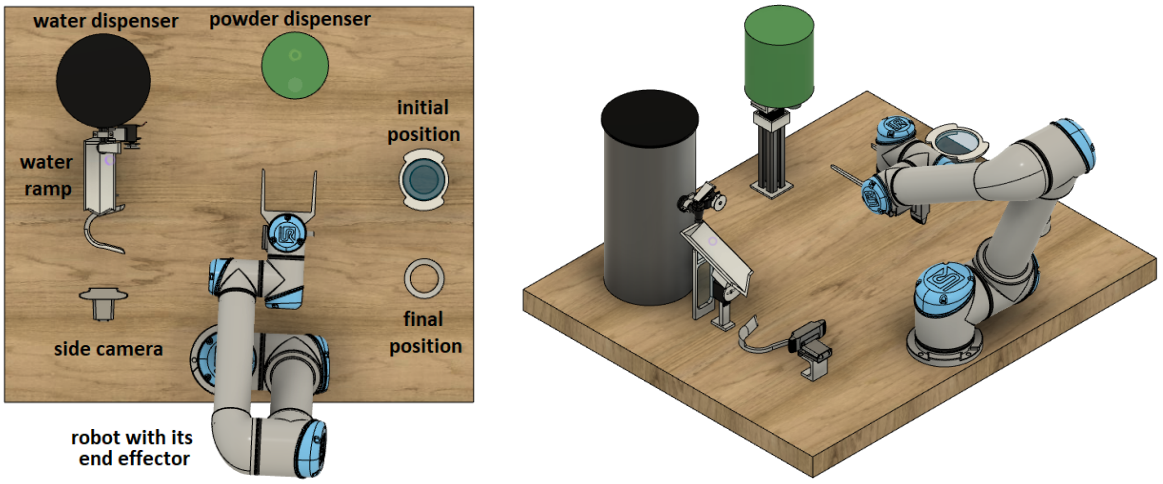
Figure 1.2 Hardware layout plan with robot path

Chapter 2

Hardware

2.1 General overview

Mother nature equipped humans with two complex manipulators and graspers in a form of arms and hands. With these tools, parallel tasks can be carried out, reducing the duration time of some processes. Engineers aim at bio-inspired inventions, some following the idea of two robotic arms cooperation in their projects [Rob]. As there was one robotic arm available for this thesis' implementation, other solutions for parallelisation and mimicking humanlike coffee preparation process were developed. Picking up the cup, placing powdered coffee inside it, stirring while pouring water and coffee evaluation, with possible further mixing, are the actions a human undertakes to prepare the drink. However, the manipulator with customised end effector serves only a purpose of a gripper, a camera holder and a stirrer, leaving a need for water and powder pouring implementation. Ready-bought boiler and powder dispenser operate purely mechanically – levers have to be pressed to release the content of the containers. They needed to be automated and added to the setup along with a controllable water ramp. Auxiliary elements for cups were designed to make them easier to pick up and to place on the table. Arduino microcontrollers, when compared to alternatives provided by STMicroelectronics or to microcomputers such as Raspberry Pi, have relatively low programming complexity (considering Arduino library), are ready to be programmed in provided IDE after connecting to them with one USB cable and have a satisfactory cost-effective ratio. Therefore, Arduino Mega (with Adafruit Motor Shield) and Arduino Uno manage the operation of respectively motors and servos in the setup. Fusion 360 software was used to model the setup components, later fabricated from PLA material with the use of 3D printers. The whole hardware system – with corresponding labels – is depicted in Fig. 2.1.



(a) Top view with labels

(b) Side view

Figure 2.1 Hardware layout

2.2 Equipment

2.2.1 Robot and its end effector

UR5 of Universal Robots product line is a 6 degrees-of-freedom collaborative robotic arm. Due to the force torque sensors incorporated in each joint, encounter of an obstacle is possible to detect and handle. Because of that, there is no enclosure separating users from the robot, which saves laboratory space and reduces maintenance costs. Additionally, when a slight misplacement in setup components occurs while running the robot script, the whole operation does not have to be terminated – user can correct the element right away without a risk of a dangerous collision with the robot. This time-saving and safe-execusion property is a major advantage when compared to non-collaborative industrial robots.

A variety of grippers are available on the market. Some of them leverage the pressure difference between environment and internal device air, some aim at utilizing servo-electric mechanisms. Bioinspired soft and adaptive grippers allow for moving delicate objects, while solid jaw-like ones may not be a suitable choice in case scuff marks are unacceptable. A decision on the end effector type depends mainly on the task characteristics. Moving a cup full of liquid is not feasible with some sorts of graspers and major adjustments would need to be implemented with regular solutions. These arguments contributed to designing a custom end effector (Fig. 2.2a, 2.2b). Two parallel claws allow for grabbing a cup and moving it to a desired location. It is a much simpler – yet sufficient – method in comparison to clamping graspers. Auxiliary holders (Sec. 2.2.5) were designed to adjust cups to the carriage method (Fig. 2.2c).

Robot's circular movement – *moveC* – operated with a Python script generated inexplicable jerks and its velocity range did not allow for imitating human's mixing speed, therefore a DC motor with attached rod acts as a stirrer. The motor is attached to the end

(a) Model (b) Real life photo (c) Real life photo with a cup

Figure 2.2 End effector

effector with two screws, a stirrer holder is clamped on its shaft. The stirring element itself is a thirteen-centimeter acrylic rod with smaller pieces, broadening its range, super-glued to the main rod. The motor is operated by *M1* pins of Adafruit motor shield mounted on Arduino Mega microcontroller (Fig. 2.3).

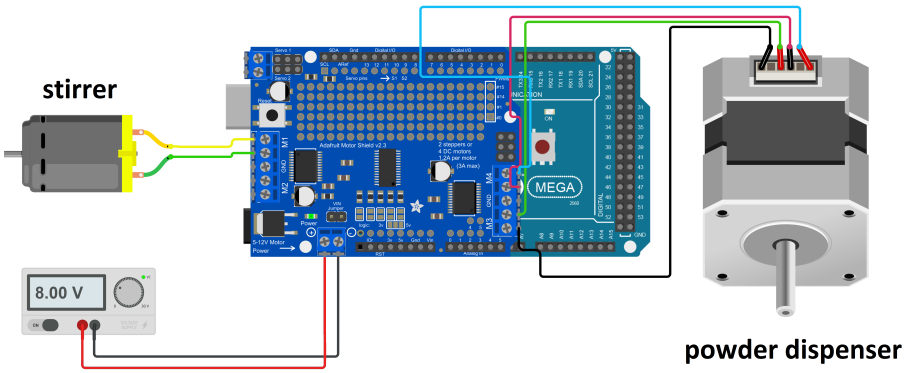
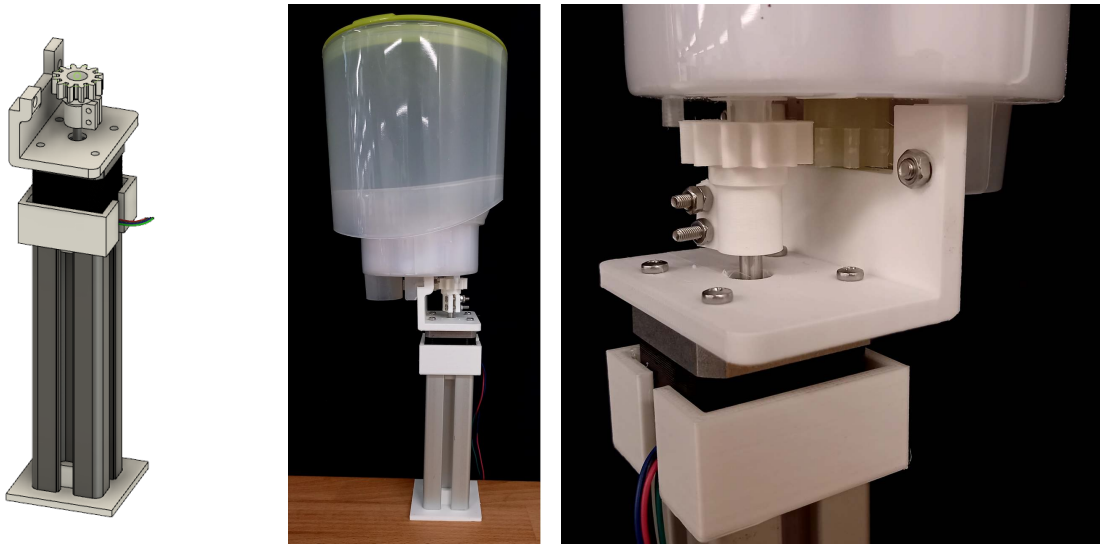


Figure 2.3 Arduino Mega with Adafruit Motor Shield pinout

Logitech C930C web camera, with the resolution of 1080x1920 px, is mounted on the flat side surface of the end effector. After mixing, the robot needs only to move to the side and slightly descend to take a photo from the top. It was important to remember that for this specific device, after running a script sending a request to the web camera to capture the image, a couple of frames need to be skipped before obtaining a sharp photo. To reduce the water steam influence on the photos, the lense was covered with a special anti-fog coating, however it alone could not eliminate the blur effect – software solutions, explained in Sect. 3.2.3, needed to be added.

2.2.2 Powder dispenser

Cappuccino powder consists of hydrophilic elements easily absorbing humidity [BMS09], therefore it was essential to store the powder in a sealed container. Such a ready-bought container with a dispensing system was supposed to serve a purpose of pouring the powder into the cup, however the initial idea of the robot pushing the lever turned out to be inefficient. Instead, the dispenser was disassembled and rearranged (Fig. 2.4). A V-slot aluminum extrusion profile with a standing support holds a stepper motor with a customized gear on its shaft. The gear affects the movement of another gear attached to a rotating platform with dosage containers. It was empirically determined that thirty-three motor steps result in one dose distribution into the dispensing hole, under which a cup needs to be placed. The motor, working with 200 rounds per minute speed, is operated by *M3*, *M4* pins on Adafruit Motorshield connected to Arduino Mega microcontroller (Fig. 2.3).



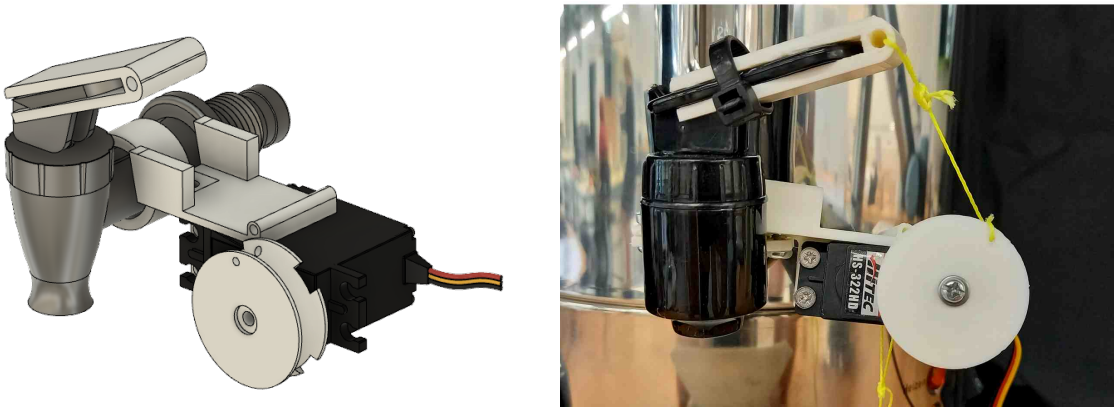
(a) Model (b) Real life photo (c) Real life photo zoomed

Figure 2.4 Powder dispenser

2.2.3 Water dispenser

According to Nescafe recommendations, water should have a temperature of 85^oC for cappuccino preparation. A commercially available boiler with temperature regulation had to be adjusted for the automatic water pouring (Fig. 2.5). Lever opening the valve got covered with an overlay with a guide hole, through which a nylon cord is passed. The cord is attached to a pulley screwed to a servo horn. A special mounting element holds the servo on the stub pipe. However, when running multiple experiments with higher temperature of water flowing through the valve, screws get looser and the mount does not keep the servo in fixed position, even with motion limiters at the top part. Unintended servo movement affects the volume of water poured into the cup. Therefore, adding a cord

guide with the cord attached to the bottom part of the boiler allowed for stabilising the device.



(a) Model (b) Real life photo

Figure 2.5 Water dispenser

Overall, the water dispenser is controlled by PWM on Arduino Uno's pin 9 (Fig. 2.6) with a simple principle of operation. The servo is in its *off* mode by default and is set to *on* mode (with specified position) for a desired time interval. In the case of this setup, the valve is closed for the value of 0 and open for position of 180.

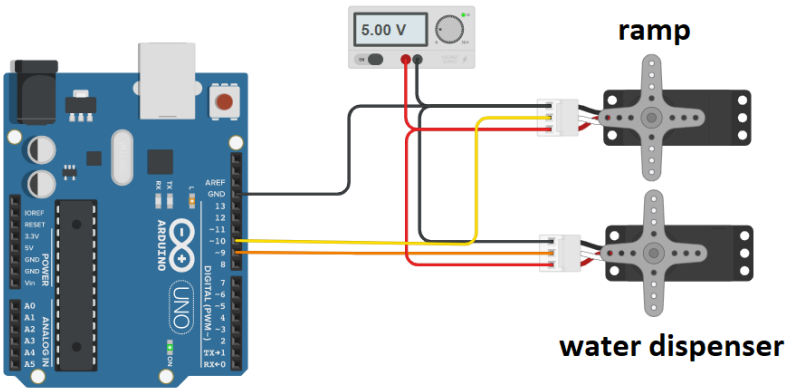


Figure 2.6 Arduino Uno pinout

2.2.4 Ramp

There were several ideas for creating an automated ramp. One of them aimed at placing a pinion on the stepper motor and composing a rack into the mobile part of the ramp (Fig. 2.7). Motor's operation would result in channel's movement along the direction pointed by the spout. This approach was abandoned due to the complexity of printing such elements and challenges regarding the design of smoothly cooperating rack and pinion.

A much simpler solution was chosen – a cam is attached to the servo horn, which, when the shaft rotates, raises the channel on distancers, adjusting the position of its spout

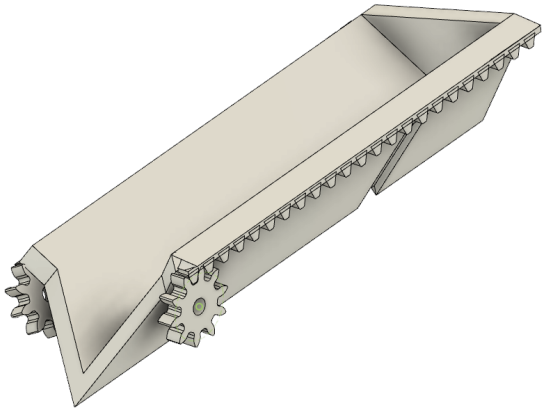


Figure 2.7 Initial idea of automating the ramp

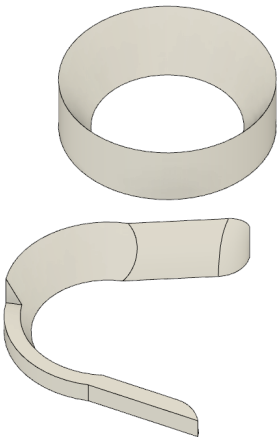
(Fig. 2.8). The first element, channel, is a composition of two perpendicular partitions with a spout and a back partition preventing water from flowing to the opposite direction. Back distancer allows for free motion (pitch) of the channel mounted to it – it is attached to the vertical column (supported with a cuboid element) with screws. The front distancer is much wider due to the fact that the servo-operated cam moves over it. Servo, mounted on a stand, is connected to Arduino Uno on pin 10 (Fig. 2.6) and operated with PWM signal. In the case if this setup, the ramp is in the highest position for 0 and in its minimal height for 180, which corresponds to a height difference of about 2 centimeters.

(a) Model (b) Real life photo

Figure 2.8 Ramp

2.2.5 Cup rim and positioners

Glass cups used in the project do not have handles, therefore designed rims (Fig. 2.9b, 2.9c) glued to the top part of cups make it easy for the robot to move them to specified destinations. Special placers ensure accurate positioning in case the element moves within end effector's claws. Two positioning rings (for initial and final coordinates) have a circular base becoming narrower at the bottom (Fig. 2.9a, 2.9c) – it makes the object slide to the centre when putting it from above. The third positioner (Fig. 2.9a, 2.9d) was designed with an idea of a cup being moved from the side to the destination coordinates. It is due to the fact that initially the ramp height was not adjustable – the spout needed to be higher than the cup but low enough to avoid collision with the end effector when stirring. Therefore, the object could not be put from above the spot and the desing had to allow for misplacement elimination. Additionally, one side of the element is trimmed to specified height to make the contents of the cup visible for the side camera.



(a) Cup positioners - models

(b) Cup rim - model

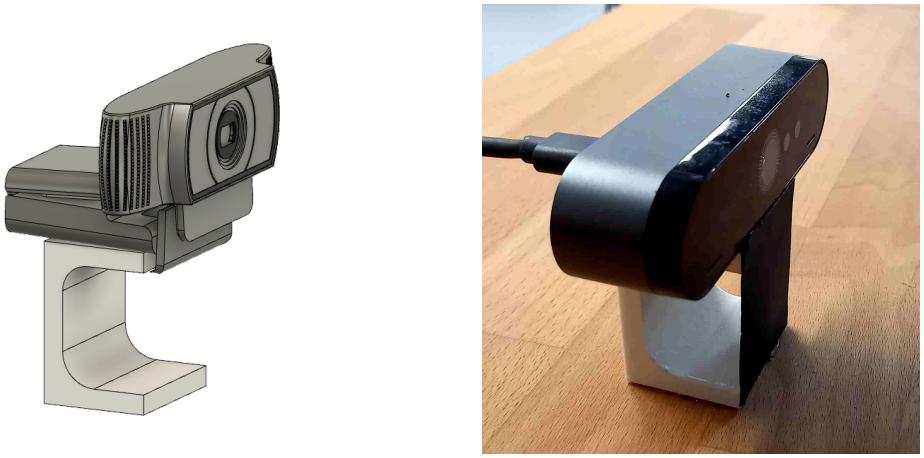
(c) Circular positioner with a cup

(d) Side positioner with a cup

Figure 2.9 Cup rim and positioners

2.2.6 Side camera

Logitech BRIO camera allows for taking full HD images of the side of the cup, used for foam height measurement and clumped powder detection. The camera is screwed to a customized stand (Fig. 2.10). As the stand was printed from white filament, its reflection on the glass cup had negative impact on image analysis. A piece of black cardboard glued to the front of the mount prevents bright reflections from affecting the results.



(a) Model (b) Real life photo

Figure 2.10 Side camera

Chapter 3

Software

3.1 General overview

Software system of the project consists of three components: Robot Operating System pipeline, computer vision aspect and bash scripts with microcontroller programs. As the setup is composed of several devices, each of them to be programmed and controlled, the need for a data exchange and proper reaction in real time appeared. ROS2 provides architecture for straightforward nodes' management and cooperation, therefore two Python packages were developed – one with interfaces definition (*roboccino_intarfaces*) and one with the actual nodes (*roboccino*). Image processing, incorporated into one of the nodes, is performed with OpenCV library. Considering lower level, microcontrollers were programmed in C++ with corresponding Arduino libraries' methods and functions. Additionally, a couple of bash scripts and aliases were created to i.e. help the user with checking if all devices are connected or defining their device names assigned by the operating system. All the abovementioned programs can be found on a Gitlab repository [Szy21].

3.2 Robot Operating System pipeline

3.2.1 ROS1 vs ROS2

ROS was created to avoid reinventing the wheel each time a robotic software is developed. Its publisher-subscriber principle of operation could be substituted with separate programs utilizing e.g. MQTT protocol, however implementing everything from the very beginning, with lots of debugging on the way, would result in unnecessary time and effort consumption. Using a provided architecture, with a robotic engineers' support community and a number of tutorials, contributed to creation of two custom *roboccino* packages, which are relatively easy to understand and reuse for people familiar with ROS.

The question that appeared after deciding to use Robot Operating System concerned the choice of ROS version. ROS1 has the advantage of being on the market since 2007, with lots of stable packages well-tested and maintained. Much more people have experience with its capabilities and issues, in contrary to ROS2, which had its first release in 2017. The aim of developing ROS2 was to make major changes in ROS architecture and bug fixes, too

many to call it another distribution of ROS1. ROS master elimination, using a different build system and targeted operating systems are just a few novelties added in ROS2. The key differences were compared in Tab. 3.1 and eventually ROS2 was selected, mainly due to its development prospects and launch file implementation in Python. The distribution used in this project is the latest one (as for summer 2021) – Galactic Geochelone.

ROS1	**ROS2**
widely known for a long time	relatively new
last release supported until 2025	supported at least until 2029
a lot of tools adjusted for ROS	tools still being transferred from ROS to ROS2
API not necessarily the same between roscpp and rospy	one base library (rcl) with similar API between rclcpp and rclpy (easy to develop rcljava, rclnodejs etc.)
initially targeting C++98	initially targeting C++11 and C++14
initially targeting Python2	initially targeting Python3
freedom in node implementation	modular structure for writing a node
launch files in XML	launch files in Python
need to run the ROS master	nodes are independent and not tight to a global master
parameters handled by the parameter server	node declares and manages its own parameters
synchronous services	asynchronous services
catkin build system	ament build system with colcon tool
Ubuntu as a main OS target	Ubuntu, MacOS and Windows 10 as OS targets

Table 3.1 Key differences between ROS1 and ROS2

3.2.2 Roboccino interfaces package

A separate supplementary package *roboccino_interfaces* provides three message types definitions utilized in *roboccino* package. The messages are associated with three devices: stirrer, powder dispenser and water dispenser. They needed to be defined so that the parameters of certain actions could be adjusted during the pipeline running. Interfaces can be summed up with Tab. 3.2 with a remark, that all fields are of type *uint8*.

Message type name	Field	Meaning
PowderDispenserParams	rpm	rounds per minute (of a stepper motor)
	steps	number of steps (of a stepper motor)
WaterDispenserParams	on_value	servo position of an open valve
	off_value	servo position of a closed valve
	time	time (in seconds) for opening the valve
StirringParams	time	stirring time (in seconds)
	speed	stirring speed (in range 0-100%)

Table 3.2 Roboccino intarfaces – message types

3.2.3 Roboccino package

Roboccino package consists of ten defined node classes, supplementary enumerated types, image processing functions (explained further in computer vision section) and launch files along with parameter assignments. The main idea is to carry out a sequence of actions presented in the diagram in Fig. 3.1. However, depending on the parameter passed to the launch file while running it, there are several modes of operation:

- **full_closed_loop** – the whole sequence with closed loop control is carried out,
- **full_single_experiment** – the whole sequence is followed excluding the closed loop actions (for open loop experiments),
- **short_closed_loop** – closed loop case starting from the situation in which a cup is ready to be filled with water and finishing the run after preparing a satisfactory coffee or reaching an ending condition (for quick testing purposes).
- **short_single_experiment** – open loop case starting from the situation in which a cup is ready to be filled with water and finishing the run after receiving data from both cameras (for quick testing purposes).

Three YAML files contain parameters' values:

- **analysis_params** – parameters for bubble and clump detectors as well as for foam height measurement,
- **hardware_params** – parameters corresponding to hardware elements (dispensers, cameras, robot, servo and motor shield commanders),
- **optimization_params** – ramp and stirrer parameters (i.e. inputs for optimization procedures).

The content of these files is to be changed accordingly to the planned task. Device names assigned by the operating system or the cameras' focus should be adjusted once, right after connecting all pieces of apparatus to the computer. Setting the optimization

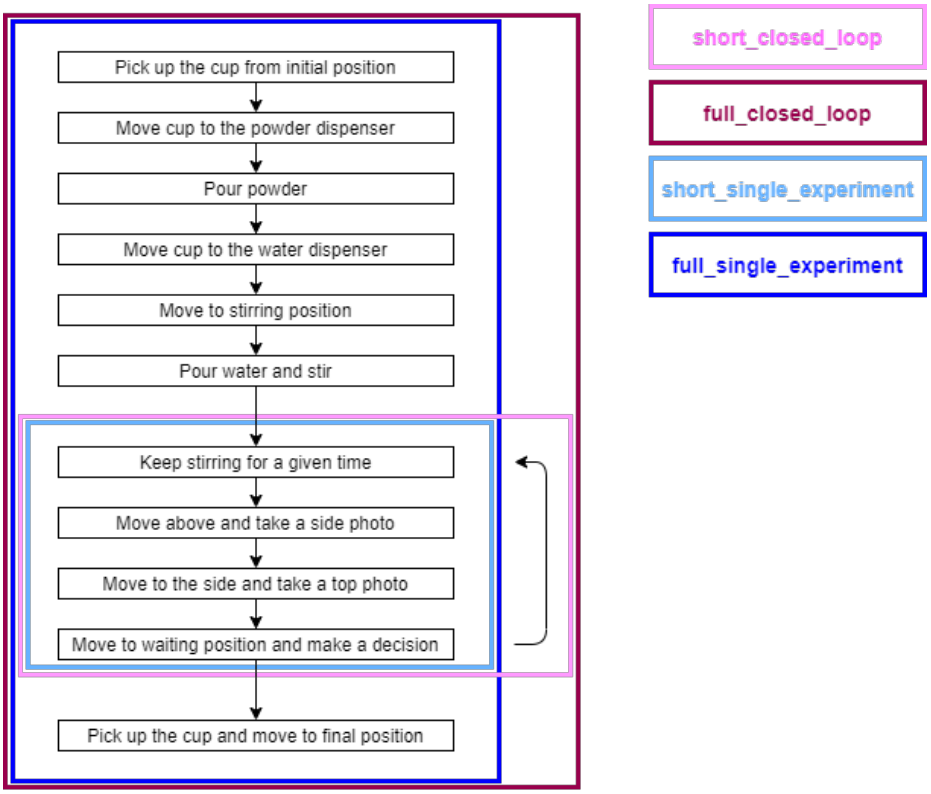


Figure 3.1 Modes of operation

parameters is done after every run of the pipeline, as specfied by the output of the optimization scripts, or once if a series of coffees prepared with the same parameters needs to be performed. The actual value assignments can be seen in the repository.

The general idea of the proposed software system design is that the *MainController* node receives information about other nodes' status and publishes suitable commands, coordinating the operation of hardware elements and image analysis. Both commands and statuses are of enum type (casted to 8-bit integers when passing over a topic as enum cannot be a message type in Galactic), to make it easier for debugging and understanding purposes. Hence, every node class has a corresponding enumerator class with fields indicating stages of operation. The main controller sends a command with the stage name and it is considered as finished when a node responds with the same message. All of the classes inherit properties after *Node* class from ROS Client Library for Python (rclpy) as well as have an implemented emergency handling. Every node is both a subscriber and a publisher to *error_signal* topic. If one of the nodes is about to get killed, launched improperly or an exception appears, it sends an error signal (of a corresponding *ErrorHandling* enum value) so that other running programs finish their operation with a proper log message printed in the console. This way a situation, in which the pipeline is running without all the devices functioning properly, will not be let to happen.

Eleven nodes are needed for the system to function, two of which are of the same class type with remapped topics, therefore a total of ten classes – explained in detail in the following subsections – are provided by the *roboccino* package.

MainController

Subscription to all status-like topics and publishing the commands to all hardware-related nodes make the *MainController* the most busy and essential element of the pipeline (Fig. 3.3). The diagram in Fig. 3.2 best describes the steps made by the controller when running.

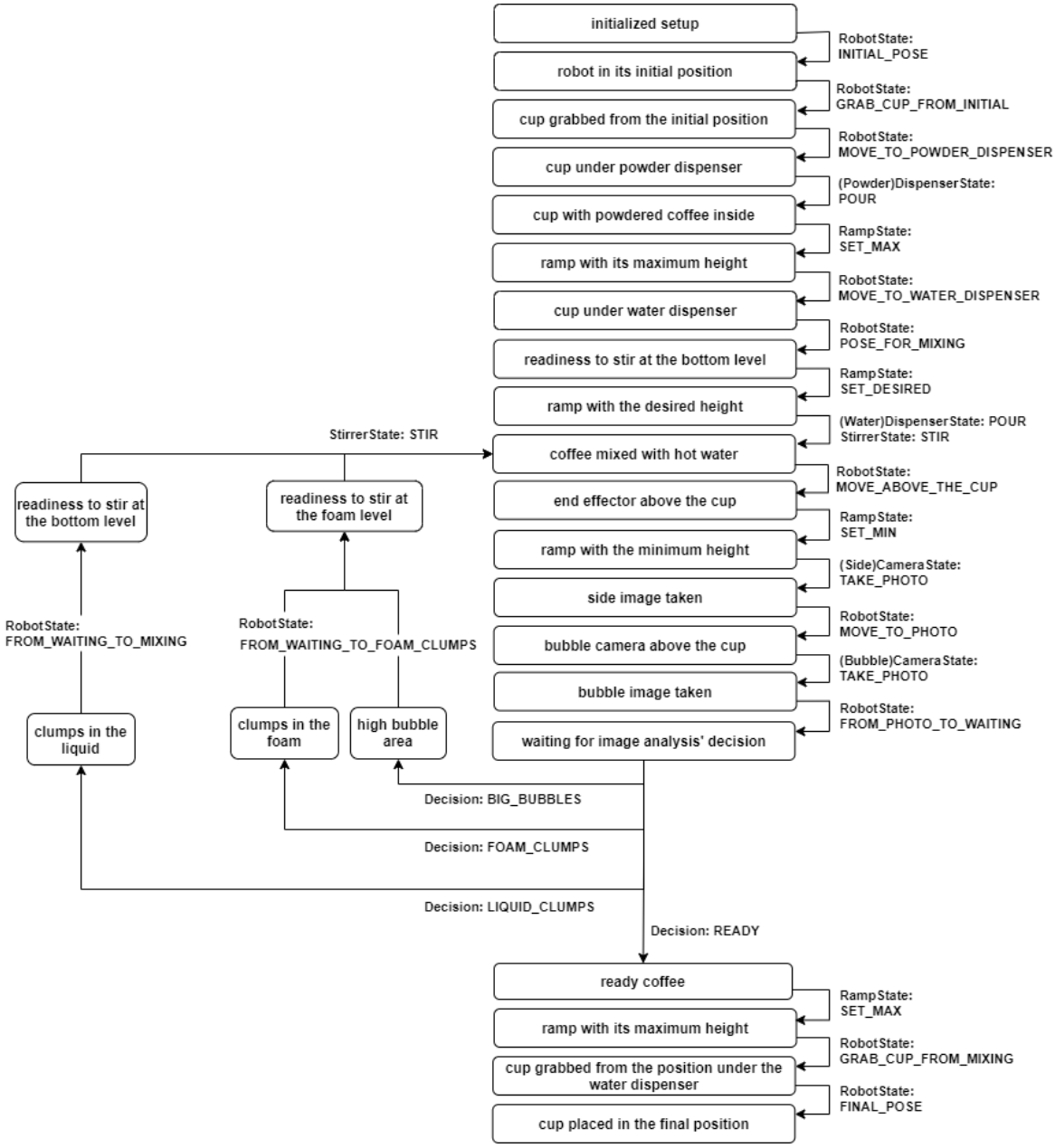


Figure 3.2 Diagram of the main controller operation with messages sent to the nodes and the states of the program

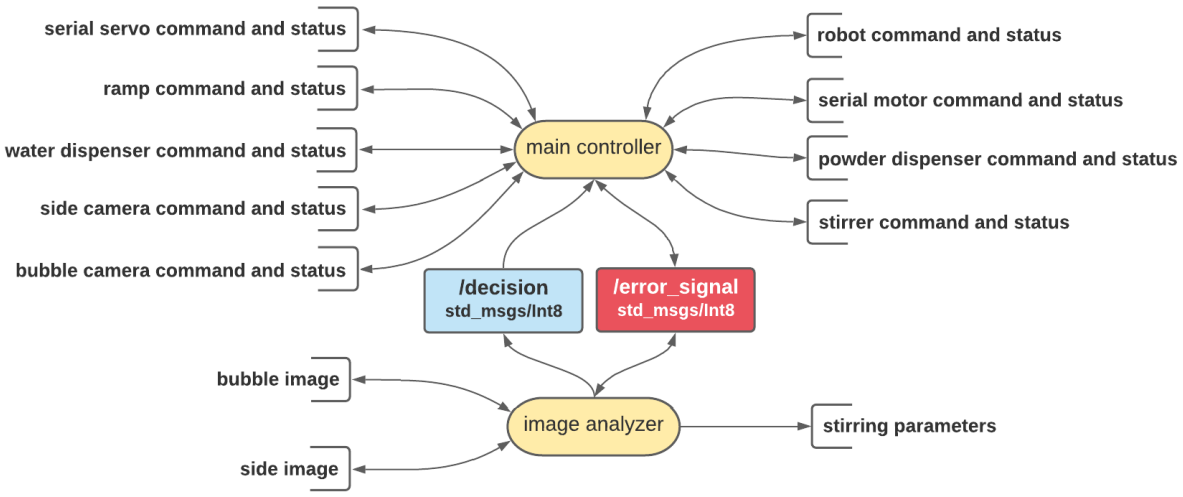


Figure 3.3 Roboccino architecture

The node has only two parameters (Tab. 3.3) – mode of operation and maximum number of iterations.

Parameter name	Default value	Description
mode_of_operation	single_experiment	mode defining the goal of the pipeline
max_iter	0	number of maximal iterations of coffee improvement attempts

Table 3.3 MainController node parameters

The system is shut down if one of following happens:

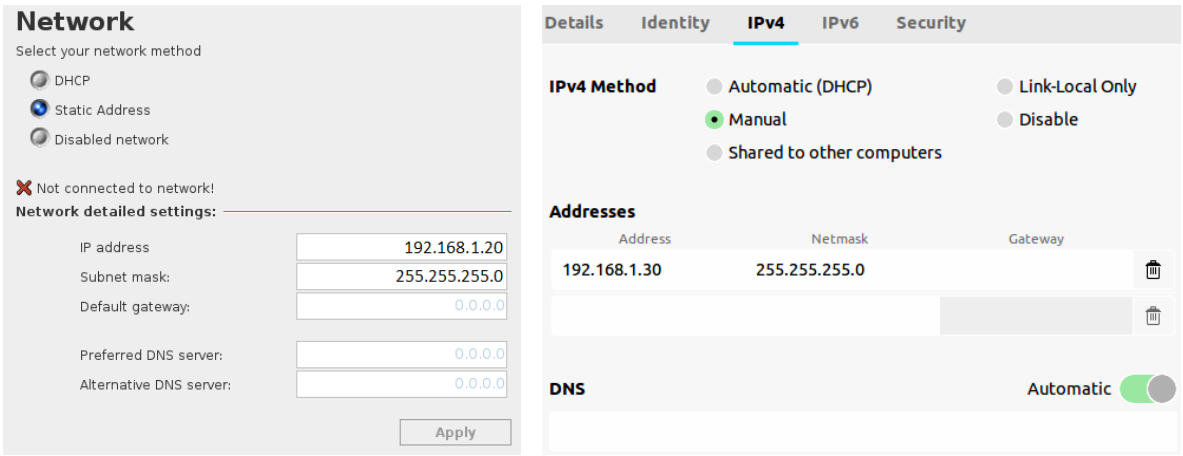
- the system finishes its task according to the chosen mode of operation,
- any of the nodes stops working properly (throws an exception or is initialized incorrectly),
- SIGINT signal is sent from the computer (then nodes send the message on *error_signal* topic).

In terms of closed loop control, the coffee is served as ready if the image analysis stated it to be without clumps and with bubble area below threshold or if the number of iterations of coffee improvement exceeds its limit.

Robot

Robot node is responsible for managing UR5 robotic arm. The operation principle of this control is based on the Real-Time Data Exchange interface allowing for external application synchronization with UR robots over TCP/IP connection. Python *ur_rtde*

API provides three interfaces, of which only *Control* interface was utilized in the node. The robotic arm's tasks consist only of moving it linearly in tool-space or joint-space to the defined positions, as stirring, dispensing powder or water is controlled by external tools – therefore there is no need to use a circular motion or operate on IO registers. Carrying out a task is as simple as running a sequence of *moveL* and *moveJ_IK* methods with path points calculated in robot node's initialization step. However, it is crucial to make sure the connection is established in a proper way. On the teach pendant a static IP option has to be chosen (Fig. 3.4a) and set along with a netmask. On the computer side, if the robot is not visible after checking the output of *ifconfig*, IPv4 might need to be set manually (Fig. 3.4b).



(a) Robot side (b) Computer side

Figure 3.4 IP configuration

Robot node, apart from the *error_signal* topic, is – as suggested in the *roboccino* introduction (Sect. 3.2.3) – subscribed to *robot_command* and publishes feedback on *robot_status* topic (Fig. 3.5).

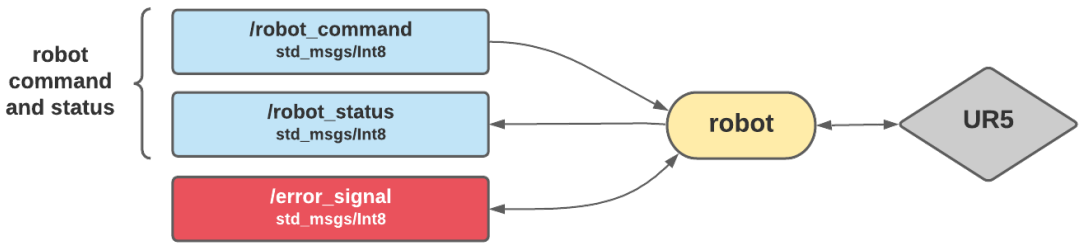


Figure 3.5 Robot node connections in ROS

To give a good overview on what the parameters of this class are, a path of the robot is visualised in Fig. 3.6. An enumerator class *RobotState* describes consecutive robot tasks bound to the path stages. Path has a predefined shape, but the key positions, on which tha path calculation is based, can be adjusted and they are passed as the robot class' parameters.

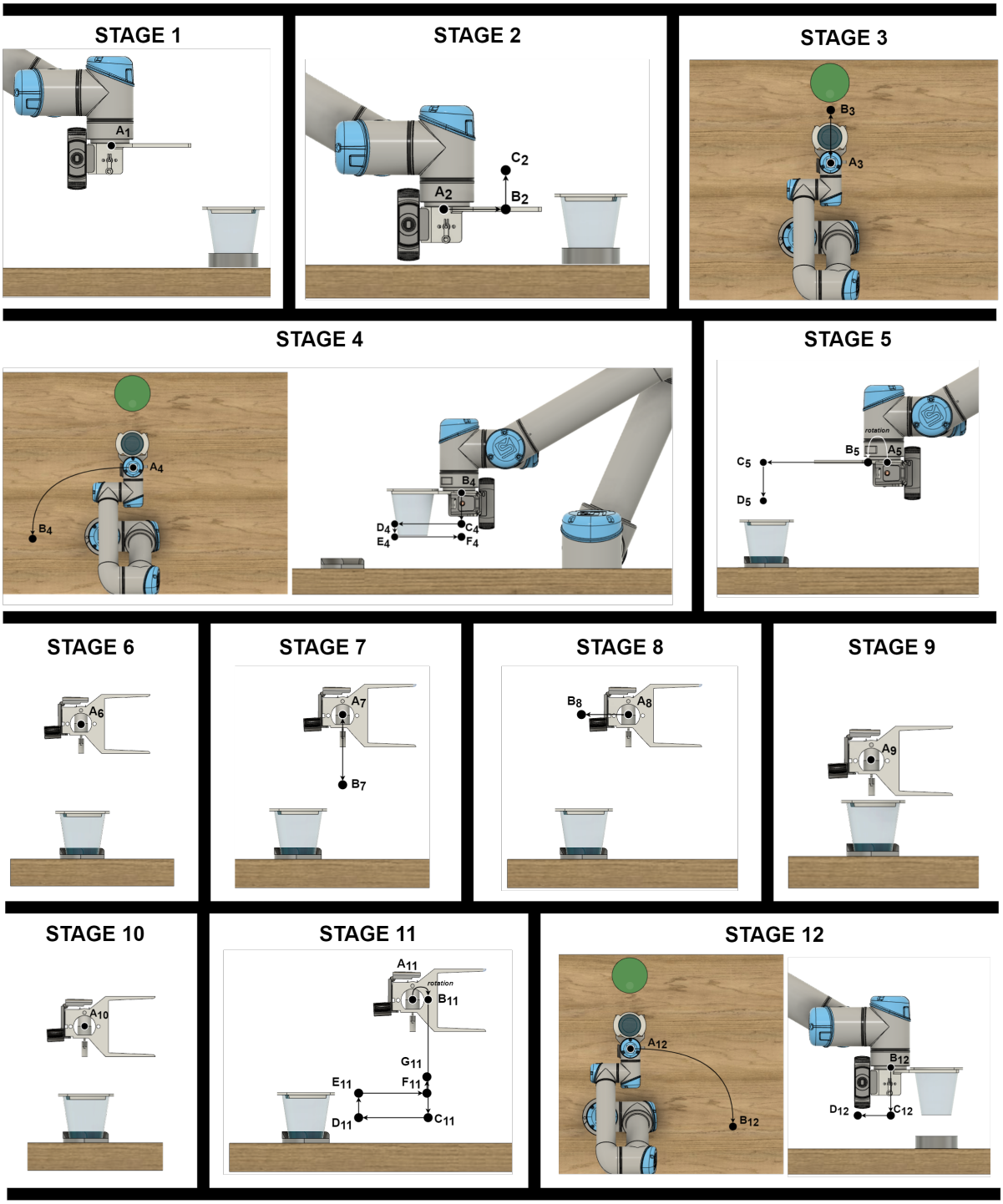


Figure 3.6 Robot path with division to stages

Regarding parametrs, their names, default values and description ars presented in Tab. 3.4. Their values used in the setup are defined in *hardware_parameters.yaml* file.

Parameter name	Default value	Description
robot_ip	192.168.1.20	IP address of the robot
velocity	0.3 m/s	robot's tool speed
acceleration	0.3 m/s^2	robot's tool acceleration
initial_cup_position	[0.0,0.0,0.0,0.0,0.0,0.0] $[x, y, z, r_x, r_y, r_z](m,rad)$	B_2: tool position in which a cup is taken from a starting point
under_powder_dispenser	[0.0,0.0,0.0,0.0,0.0,0.0] $[x, y, z, r_x, r_y, r_z](m,rad)$	B_3: tool position in which a cup is under powder dispenser valve
under_water_dispenser	[0.0,0.0,0.0,0.0,0.0,0.0] $[x, y, z, r_x, r_y, r_z](m,rad)$	E_4: tool position in which a cup is under water dispenser valve
final_position	[0.0,0.0,0.0,0.0,0.0,0.0] $[x, y, z, r_x, r_y, r_z](m,rad)$	C_{12}: tool position in which a cup is in the final point
mixing_depth	0.0 m	difference between the tool position above the cup (C_5) and the mixing position (D_5)
clump_mixing_depth	0.0 m	difference between the tool position above the cup (C_5) and clump mixing height (A_{10})

Table 3.4 Robot node parameters

Camera

There are two nodes of *Camera* type, as there are two web cameras in use in the setup – one facing the side of the cup and the other one on the end effector. Therefore the subscription to *camera_command* and publishing to *camera_state* is remapped in the launch file by adding a prefix – either *side* or *bubble*. Additionally, also *image* topic is extended with a prefix. Error handling topics stay the same in both cases (Fig. 3.7). All the node's parameters are presented in Tab. 3.5.

The camera handler is of *VideoCapture* type from OpenCV library content. A device name is needed for initialization, followed by setting the focus, motion-jpeg codec (4-character code of codec used to compress the frame in a form of 'M', 'J', 'P', 'G') and the resolution size (1920x1080 px). As cameras do not have many tasks to carry out, only two commands of *CameraState* enum are sent: initialization command and a photo taking request.

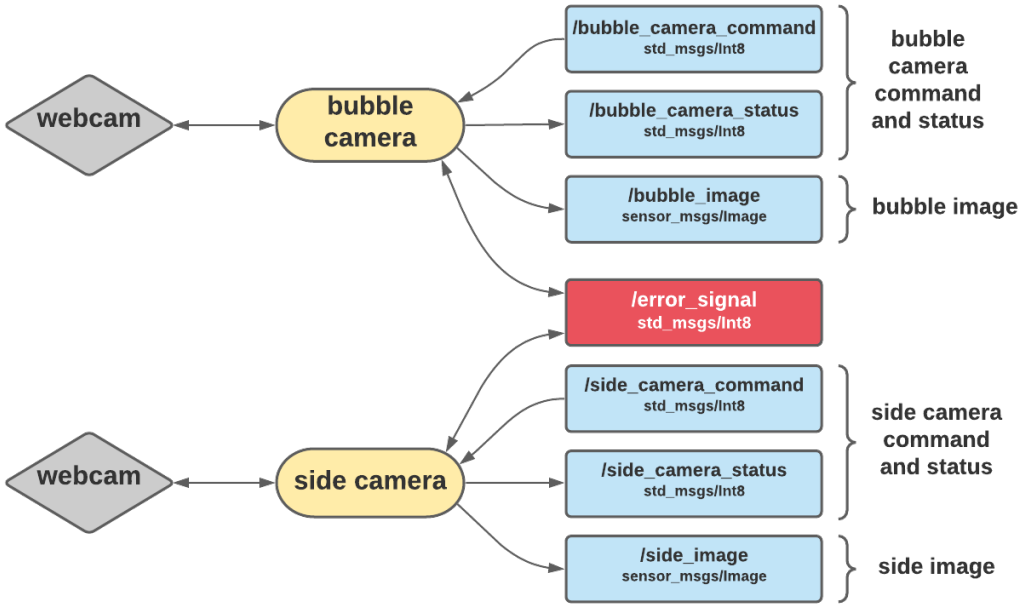


Figure 3.7 Web camera nodes connections in ROS

Parameter name	Default value	Description
device	/dev/video0	device camera name assinged by the OS
focus	0	[0; 255] value defining focus level
preprocessing_type	none	mode of preprocessing (either *side* or *bubble*)
waiting_time	0 *s*	value defining time interval of waiting between reaching the photo position and taking a photo
skipped_frames	0	number of skipped frames before saving an image
crop_x	0 px	X coordinate of the upper right pixel needed for cropping
crop_y	0 px	Y coordinate of the upper right pixel needed for cropping
crop_width	0 px	width of the cropped image
crop_height	0 px	height of the cropped image

Table 3.5 Camera node parameters

There are several steps undertaken when the *MainController* sends a *TAKE_PHOTO* command. First of all, if the device is ready, the node waits for a predefined time interval of *waiting_time*. It is necessary in the first iteration of closed loop control experiments as the water vapour over the hot drink right after pouring affects the photos – they are too blurred to properly analyze them. Next, a series of photos is captured and skipped, their number defined by *skipped_frames* parameter, eventually saving the last image. It is done

due to the fact that the utilized cameras' first frames after launching the capture mode are too noisy and devices automatically reduce the noise after a moment. The image is then saved locally as a PNG file, preprocessed in accordance to the *preprocessing_type* mode of operation, converted from OpenCV format to BGR8 and published on *image* topic. As for cropping, there are four parameters needed this step. An image can be treated as a 2D array of values, therefore cropping is performed simply as assinging a subarray to a new variable. This is performed by one line of code:

```
cropped_image = current_image[y:y+height , x:x+width]
```

where (x, y) are the coordinates of the upper left pixel of the subarray and (width, height) define the size (in pixels) of the new image.

ServoCommander

The node is responsible for the serial connection with a microcontroller handling servos corresonding to the ramp and water dispenser (Fig. 3.8). Two separate nodes exchanging data with the same microcontroller might create some packet collisions or mysterious errors on the way. Therefore, when the *MainController* sends a request to change the servo position, it is passed through the *ServoCommander* node. The PySerial library serves a purpose of connection handlers provider. Only two paramters are required to set a communication bridge – the device name and baud rate (Tab. 3.6). Their initialization happens after receiving a *SerialStatus* message on *serial_servo_command* topic. Afterwards, the confirmation of the readiness is sent to *MainController*.

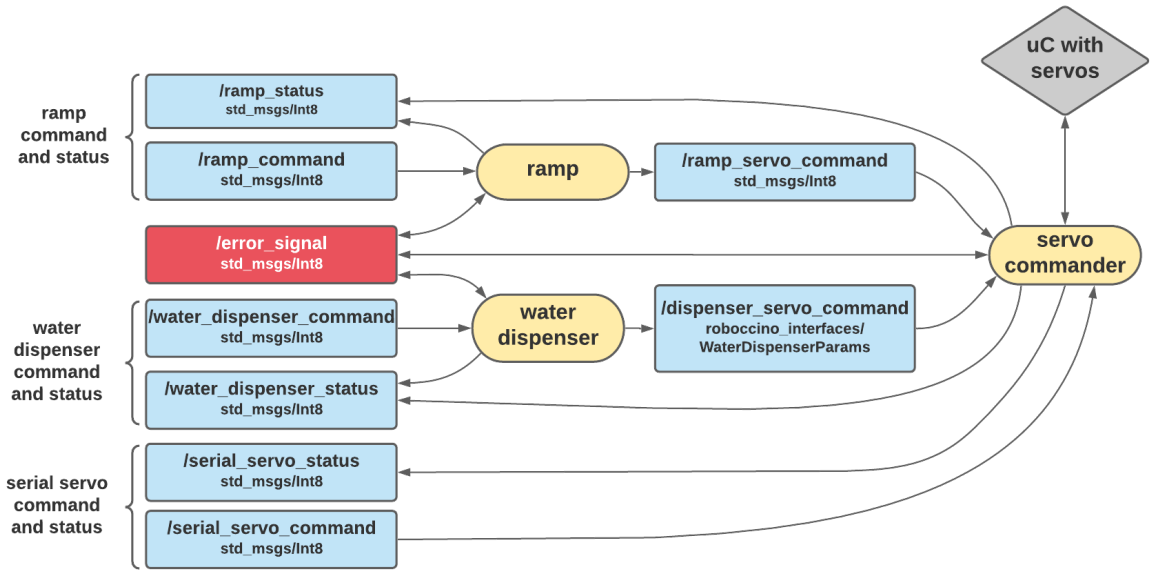


Figure 3.8 ServoCommander node with ramp and water dispenser connections in ROS

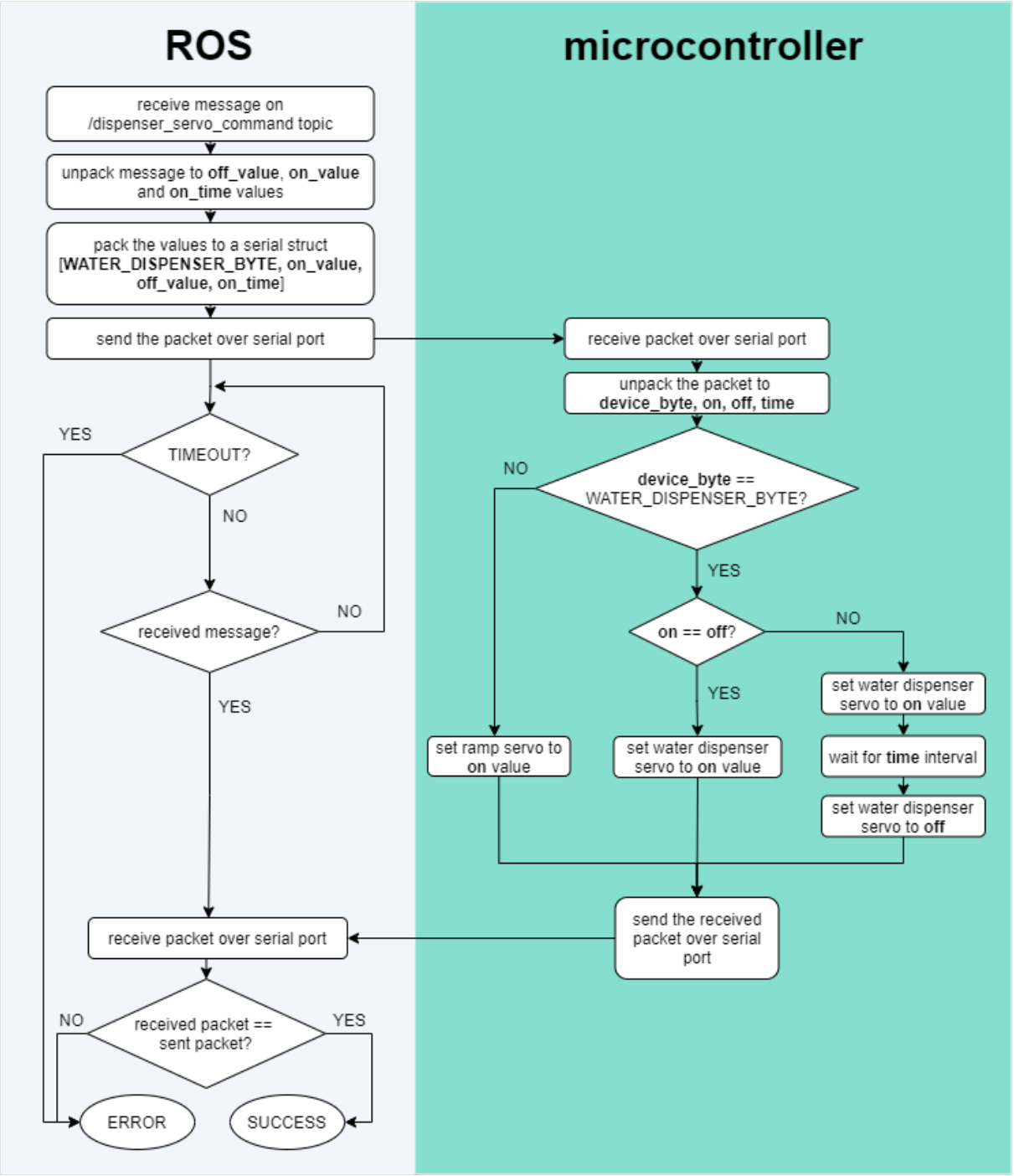


Figure 3.9 Communication diagram between ROS and the microcontroller

Parameter name	Default value	Description
device	/dev/ttyACM0	uC device name assinged by the OS
baud_rate	9600 *bits/s*	rate at which bits are sent on serial bridge

Table 3.6 ServoCommander node parameters

The procedure of sending data to the microcontroller can be easily explained on the example of the water dispenser (Fig. 3.9).

1. Desired *on* and *off* servo positions with opening time (packed in *WaterDispenserParams* message as in Sect. 3.2.2) are sent from *WaterDispenser* node to *dispenser_servo_command*, to which *ServoCommander* is subscribed.

2. Values are packed in a 4-byte packet in form of [WATER_DISPENSER_BYTE, on_value, off_value, on_time].

3. The packet is sent over the serial connection and the response is received.

4. If the response contains the same values as the desired ones, the confirmation is sent to *water_dispenser_status* topic.

Similar procedure is followed for the ramp servo, with topics changed analogously. The only real difference is in terms of messages and the data packet. *Ramp* node sends only the desired servo position of 8-bit unsigned integer type and the packet is as follows: [RAMP_BYTE, servo_position, 0, 0]. From the microcontroller implementation side, it was easier to stay with a constant size of the packet and filling bytes with zeros than to switch between two- and four-byte data. The WATER_DISPENSER_BYTE and RAMP_BYTE – set as global constants in the code – are used by the microcontroller to distinguish which servo should be operated at the moment of receiving data.

In case an exception is thrown or a message is received on *error_signal* topic or the *MainController* commands to finish the node operation, it is crucial to close the water dispenser valve before killing the node. Therefore, a proper data packet is sent just before destroying the communication socket. Setting the ramp position is not as important then, because it would not cause any damage to the setup, in contrary to constantly flowing hot water.

Ramp

Ramp has a simple implementation – apart from standard topics related to error signal as well as commands and status for *MainController* it has only one additional publisher option. When the *MainController* commands the ramp with *RampState* message to be set on some desired height, it sends the servo position on the *ramp_servo_command* topic. The node does not send feedback to the *ramp_status*, as it is done by the *ServoCommander* after confirmation that action was actually performed by microcontroller. However, the feedback is slightly different than in previous cases – it does not match the *MainController*

command, which can be either SET_MIN, SET_MAX, SET_DESIRED. Three cases are confirmed with SET status, as *ServoCommander* would need to make distinction between minimal, maximal and desired heights, which would require further implementation, not crucial for the overall operation.

The node has three parameters (Tab. 3.7), which are set in *optimization_params.yaml* file as the ramp height is one of the inputs to optimization scripts.

Parameter name	Default value	Description
desired_height	90	servo position correspodning to the ramp height desired for water pouring
max_height	0	servo position corresonding to the maximal ramp height
min_height	180	servo position corresonding to the minimal ramp height

Table 3.7 ServoCommander node parameters

WaterDispenser

Managing the water dispenser is analogical to the case of the ramp. After receiving a *DispenserState* command of POUR value, the node packs *time*, *on* and *off* servo positions to *WaterDispenserParams* message and sends it to *ServoCommander*, which is then responsible for sending feedback to *MainController*. It is important to mention that in the *hardware_params.yaml* the time, for which the valve should be open, is a double value with the accuracy of 0.5. It is multiplied by 2 and casted to integer when the parameter is imported to the node, as passing a double is harder to implement on both sides of serial communication. Microcontroller, after receiving the data packet, divides the time value by 2, this way obtaining the initial desired time.

The parameters of this node are presented in Tab. 3.8.

Parameter name	Default value	Description
time	2.5 s	time interval for which the valve should be open to fill the cup with water
on	180	servo position corresonding to the maximal ramp height
off	0	servo position corresonding to the minimal ramp height

Table 3.8 ServoCommander node parameters

MotorShieldCommander

The implementation of *MotorShieldCommander* assumes that a microcontroller managing the powder dispenser and the stirrer is connected via USB to the computer and

that serial communication can be established. As in the case of the *ServoCommander*, it did not make sense to create serial connections to the same device in separate nodes. Topics *serial_motor_command* and *serial_motor_status* connect the node to the *MainController*, while error handling is performed with standard *error_signal* (Fig. 3.10). The parameters are the same as for *ServoCommander* (Tab. 3.9), as only device name and baud rate are necessary for the communication.

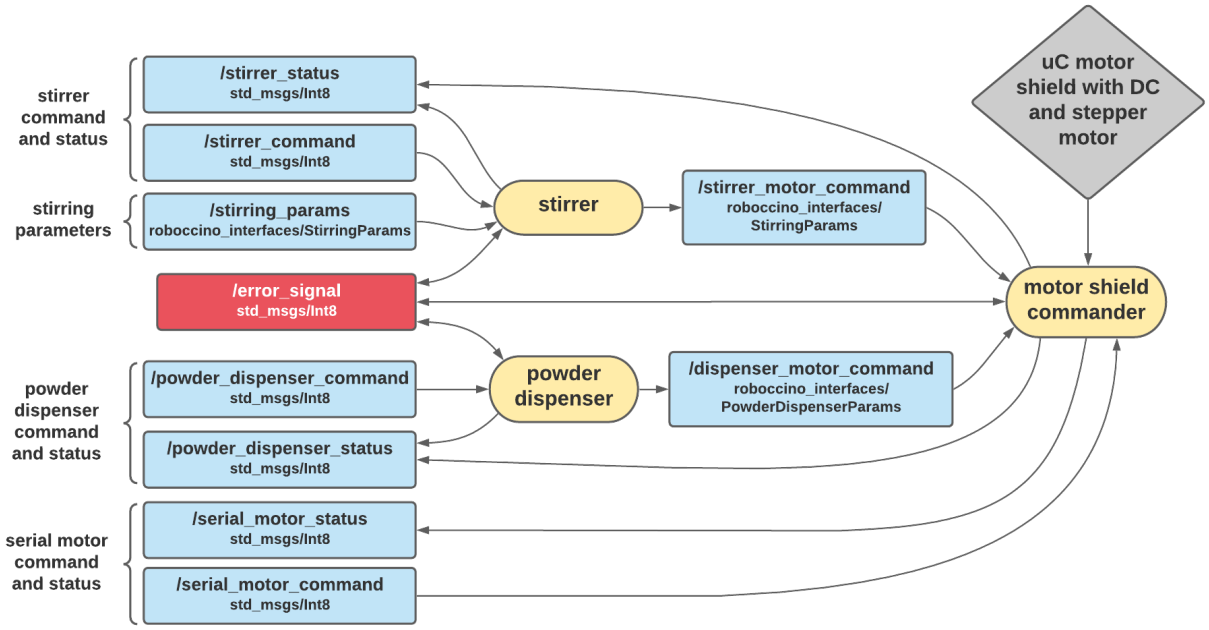


Figure 3.10 MotorShieldCommander node with stirrer and powder dispenser connections in ROS

Parameter name	Default value	Description
device	/dev/ttyACM1	uC device name assinged by the OS
baud_rate	9600 *bits/s*	rate at which bits are sent on serial bridge

Table 3.9 MotorShieldCommander node parameters

The procedure of operating a motor can be shown on an example of powder dispenser:

1. Desired *rpm* and *steps* of the stepper motor packed in *PowderDispenserParams* message (Sect. 3.2.2) are sent from *PowderDispenser* node to *dispenser_motor_command*, to which *MotorShieldCommander* is subscribed.

2. Values are packed in a 3-byte packet in form of [POWDER_DISPENSER_BYTE, rpm, steps].

3. The packet is sent over the serial connection and the response is received after the motor action is performed.

4. If the response contains the same values as the desired ones, the confirmation is sent to *powder_dispenser_status* topic.

Analogically, operating the DC motor of the stirrer is performed. Its parameters are passed in a *StirrerParams* message type and the packet is of a form of [STIRRER_BYTE, stirring_time, speed]. As the stirring speed is sent as a 0-100% value, it is here calculated to correspond to [0; 255] range. POWDER_DISPENSER_BYTE and STIRRER_BYTE are set as global constants in the code and are used by the microcontroller as a distinguisher which motor to operate.

PowderDispenser

PowderDispenser, subscribed to *powder_dispenser_command* and publishing to *owder_dispenser_status* with *DispenserState* enum messages, is responsible for sending proper information on the *dispenser_motor_command*. Its rounds per minute and number of steps parameters (Tab. 3.10) correspond to full dispensation of a portion of powdered coffee.

Parameter name	Default value	Description
rpm	10 $round/m$	stepper motor speed used to dispense powder
steps	34	number of steps required to dispense powder

Table 3.10 PowderDispenser node parameters

Stirrer

Apart from the elements typical for this system design (meaning error signal handling, *StirrerStatus* information exchange with *MainController* and passing motor operation data to the *MotorShieldCommander*), *Stirrer* needed an implementation for changing stirring parameters. The initial values of stirring time and speed (Tab. 3.11) are imported from *optimization_params.yaml* file, but during the pipeline running stirring can be performed more than once in closed loop control mode. Therefore, new parameters based on the image analysis are published on *stirring_params* topic and saved by the *Stirrer* node for further operation.

Parameter name	Default value	Description
speed	10 %	DC motor speed while stirring
time	10 s	stirring time

Table 3.11 Stirrer node parameters

ImageAnalyzer

ImageAnalyzer differs from the hardware-related nodes. First of all, it does not have a topic over which the *MainController* passes its commands and on which the status is published in response. It does, however, have the standard connection to *error_signal*. Due to the subscription to *bubble_image* and *side_image*, it has access to photos taken by the cameras.

After receiving an image on one of the topics, it is processed in accordance with the computer vision analysis described in detail in Sect. 3.3. Figures with cropped image, bubble and clump detection and bubble area analysis are saved in terms of top foam image. As for the side photo, apart from the cropped version, clumps detected in liquid and foam measurement figures are stored. Information received from *stirrer_motor_command* allows for naming the saved processed images with values of stirring time and speed as suffixes. If both side and top photos got analyzed, the decision procedure is launched. It checks several cases of the coffee state, publishes a proper message over *decision* topic and – if needed – forwards stirring parameters packed to *StirringParams* message to *stirring_params* topic. In case the decision making procedure receives incorrect inputs or an error occurs, information is sent to *error_signal* topic.

The following variables are utilized in the algorithm (cf. Alg. 1):

- T_{clumps} $[s]$ – predefined time prescaler used to calculate the new stirring time in the case of clump detection (in the current program version equal to 60 seconds),
- $T_{bubbles}$ $[s]$ – predefined time prescaler used to calculate the new stirring time in the case of high bubble coverage (in the current program version equal to 60 seconds),
- S_{clumps} [%] – predefined speed prescaler used to calculate the new stirring speed in the case of clump detection (in the current program version caluclated speed is equal to the initial speed),
- $S_{bubbles}$ [%] – predefined speed prescaler used to calculate the new stirring speed in the case of high bubble coverage (in the current program version caluclated speed is equal to the initial speed),
- thr [%] – threshold corresponding to the boundary between acceptable and unacceptable bubble coverage in the foam part (passed as a node parameter),
- a_{liquid} [%] – area occupied by clumped powder in the liquid part,
- a_{foam} [%] – area occupied by clumped powder in the foam part,
- $a_{bubbles}$ [%] – area occupied by bubbles in the foam part,
- d – variable indicting the decision, of *Decision* enum type.

Almost all parameters of this node, placed in *analysis_params.yaml*, refer to the image analysis, therefore their meaning is explained in Sect. 3.3. The only parameter participating in the decision making procedure is *good_coffee_threshold*. Its value defines the

maximum area occupied by the bubbles in the top image that is still considered as a determinant of a satisfactory coffee.

Algorithm 1 An algorithm of decision making by the ImageAnalyzer node

Require: $T_{clumps} > 0$, $T_{bubbles} > 0$, $S_{clumps} > 0$, $S_{liquid} > 0$, $a_{liquid} \geq 0$, $a_{foam} \geq 0$, $a_{bubbles} \geq 0$, $thr > 0$

```
if a_liquid > 0% then
    d ← LIQUID_CLUMPS
    t_stirring ← T_clumps · a_liquid/100%
    s_stirring ← S_clumps · a_liquid/100%
else if a_foam > 0% then
    d ← FOAM_CLUMPS
    t_stirring ← T_clumps · a_foam/100%
    s_stirring ← S_clumps · a_foam/100%
else if a_bubbles ≤ thr then
    d ← READY
else if a_bubbles > thr then
    d ← BIG_BUBBLES
    t_stirring ← T_bubbles · a_bubbles/100%
    s_stirring ← S_bubbles · a_bubbles/100%
end if
```

3.3 Computer vision

Human eyes combined with the analysis performed in the brain are a highly complex real-time system giving quick feedback on the surroundings and allowing for proper reaction. Scientists aim at making machines see the world as people do and respond to the occuring situations. Computer vision [Hua96], also reffered to as CV, is the field, whose development made it possible for the computers to mimic human vision by extracting some high-level information of the environment from digital images [Ros88]. Some of the tasks carried out by the CV are object detection [Zou19], classification [DKD15, GL10], motion analysis [CL08, CBK+21] or image restoration [XWD13]. By applying several transformations to an array of pixels, complex real world scene is changed into a numerical value (e.g. object sizes) or a symbolic information (e.g. shape or texture of the elements). There are a lot of challenges facing this scientific field [Zha10] – analysis of high resolution images takes a lot of computation time and resources, while humans perform it unconsciously, without much focus – therefore new methods are constantly looked for and developed.

In this project the robotic system needs feedback on the coffee quality. The development of the data extraction on each of these aspects – mainly in forms of detections and classifications – is explained further in this section. Initially, five factors were taken into consideration as determinants of good coffee – presence of powder clumps in the liquid, presence of powder clumps in the foam, foam height, number and sizes of bubbles.

It is crucial to remember that image analysis is highly depenedant on the light distribution in the room. To achieve high repeatability in running multiple experiments, the setup was covered with black fabric and a light ring with dispersion cover was placed over the photo taking area. Different light colours were tested (Fig. 3.11), but the most important aspect of the light was to make it uniformly distributed rather than to have a specific color. When trying to disperse the light source, it turned out that the output colors were not intensive enough for the photos to differ much, therefore the white light was chosen due to the higher availability on the market, especially considering possible research continuation.

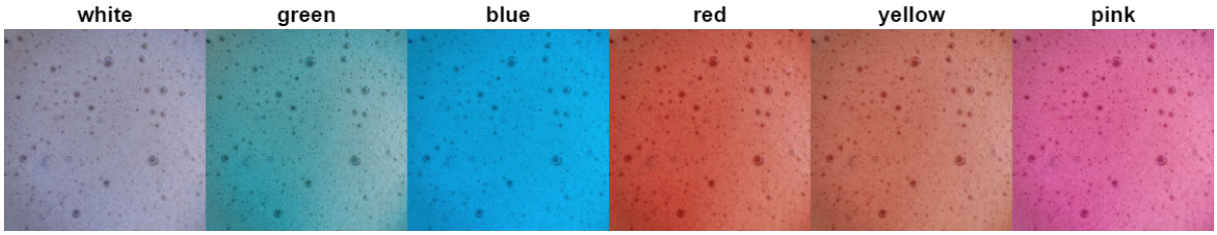


Figure 3.11 Artificial light colors test

3.3.1 OpenCV

OpenCV is an open-source library providing tools for real-time optimized Computer Vision. It supports a variety of programming languages, so its Python implementation was used in the system. The library has a community support, plenty of introductory tutorials, is well integrated with many Python tools and is commonly used in computer vision applications. An attempt to utilize Matlab CV tools was initially made, however calling its functions in Python with Matlab Enginge API consumes more computational time when running [MHB12]. Additionally, OpenCV has the advantage of being free, while Mathworks software requires a license. Taking all into conisderation, OpenCV seemed to be a reasonable software package and was eventually chosen as a basis for coffee analysis (with one function taken from scikit-image library).

3.3.2 Bubble detection

Bubbles are detected on images taken from the top view of the cup. The resolution of these photos is 1920x1080 px. Various tools and transformations were examined and compared before determining the final detector.

Initial ideas

The initial approach was to detect the number and sizes of bubbles as circles in the foam. Before realising how much time Matlab consumes for simple operations and how

little of parameter tuning can be performed, a few iterations of testing its functions was done (Fig. 3.12).

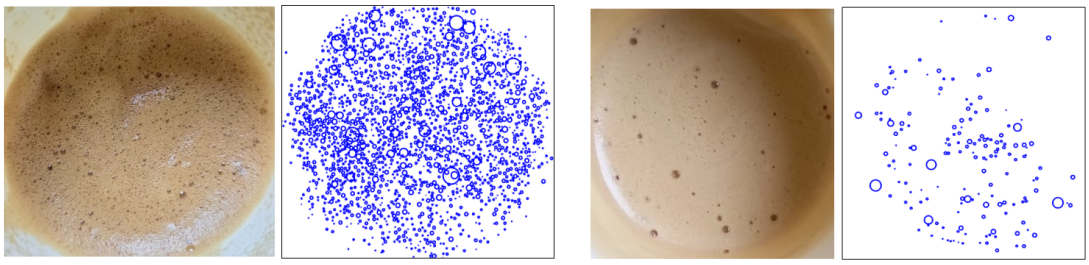


Figure 3.12 Matlab circle detection functions for coffee foams of low and high quality

Circles detection was later implemented with OpenCV. Tested Hough circle transform with Hough gradient method (Fig. 3.13) is in general not complicated to apply, however it has two parameters (gradient value used to handle edge detection and accumulator threshold value for the Hough gradient method) that turned out to be difficult to tune for images.

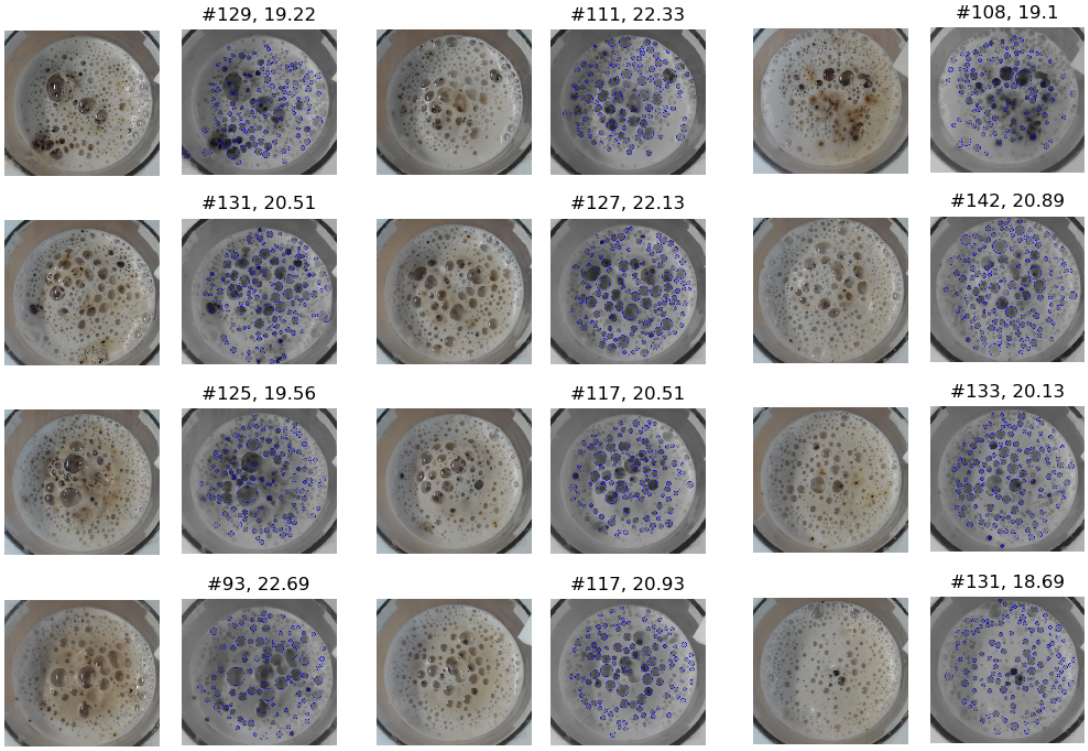


Figure 3.13 Parameter tuning for Hough circle detection – testing different parameters configurations with the results in a form of pairs <input image, circle detection> with displayed detected circles' number and mean diameter in pixels

One of the possible reasons of not detecting all bubbles was the fact that not every bubble has a circular shape – some of them are deformed or more eliptic. Hence, the blob detection was investigated. In OpenCV – apart from area and thresholds – circularity,

inertia and convexity can be adjusted, making it a very good detector for bubbles, which in fact can be seen as dark blobs in the foam. This approach was eventually decided to have the best and the most accurate results, so it was used in the final detector. Instead of number of bubbles and their sizes, the overall area occupied by the blobs is the determinant of the readiness of the coffee.

Artificial neural networks, so widely applied to lots of computer vision problems nowadays, were considered as an alternative way of analysing coffee quality. Supervised learning however would require a training dataset – no ready and relatable datasets were found on the internet and making one would result in lots of time spent on labelling. Moreover, the task would not be a simple classification problem as feedback on whether the coffee is ready or not is not enough – the information on the quality level is necessary. Unsupervised or reinforcement forms of learning would also be problematic to implement or insufficient in terms of their feedback, therefore the idea of using neural networks was abandoned.

Final detector

The final detector is a combination of three stages of blob detection. Different transforms in various combinations – by trial and error – were tested while seeking for the efficient sequence with tuned parameters and the image analysis pipeline presented in Fig. 3.14 gave the best results.

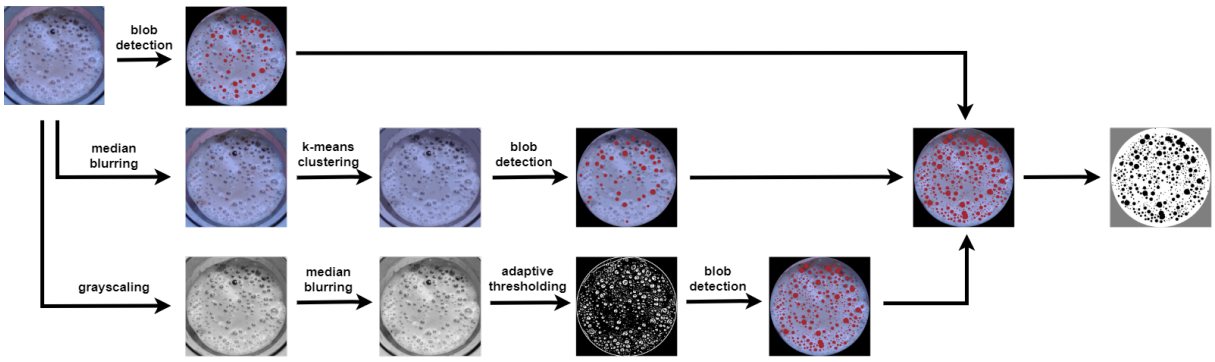


Figure 3.14 Bubble detection pipeline

The first subdetector takes the cropped image and performs the blob detection with paramters of Detector 1 in Tab. 3.12. The second subdetector in the first place applies a median blurring with kernel size equal to 7, performs K-means clustering with $k = 8$ and then applies a blob detection with values of Detector 2 in Tab. 3.12.

The last stage of blob detection starts with converting image into grey scale. After median blurring with k=7, adaptive thresholding is applied with the following properties:

- if pixels are above the therhold value, they are assigned with 255 value,

- binary thresholding with threshold value as the mean of neighbourhood area (size of pixel neighbourhood equal to 11 and constant substracted from the mean as 2).

Parameter name	Detector 1	Detector 2
minimum threshold	10	10
maximum threshold	200	200
minimum convexity	90%	90%
maximum convexity	100%	100%
minimum area	10 px	400 px
maximum area	115200 px	115200 px

Table 3.12 Blob detectors' parameters

In contrary to previous cases, blob are found with a function from scikit-image library. Difference of Gaussian [Sci] method with the maximum standard deviation for Gaussian kernel equal to 20 and a lower bound for scale space maxima of 0.5 value were chosen for this specific case.

After all three detections are done, they are merged and converted to an image with white background and grey mask, on which black pixels indicate the presence of the blob. This way, even if the same bubbles were detected by all three methods, the repetition does not affect the result – black pixels cover the same area. The ratio of black pixels to white pixels is calculated and changed to percentage data, which is later passed to decision procedure and optimization scripts.

3.3.3 Foam height measurement

Measuring the foam height was eventually not considered in closed-loop control and optimization. The main reason for that was small diversity of heights – coffees of both satisfactory and poor quality had a similar-looking side foam photographs. However, as it might change in further research, the computer vision pipeline (Fig. 3.15) was developed for this purpose.

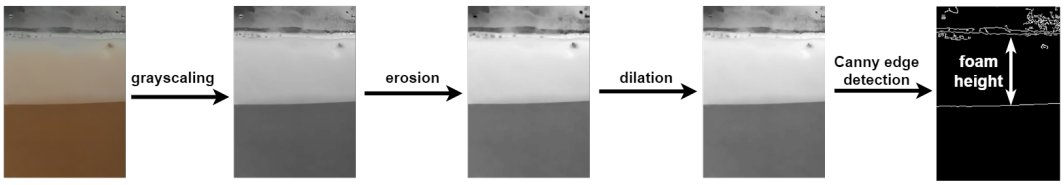


Figure 3.15 Foam height measurement pipeline

Several attempts of method adjustment were performed (Fig. 3.16), also in terms of determining the most reliable statistical indicator of the foam height, mainly considering mean and mode.

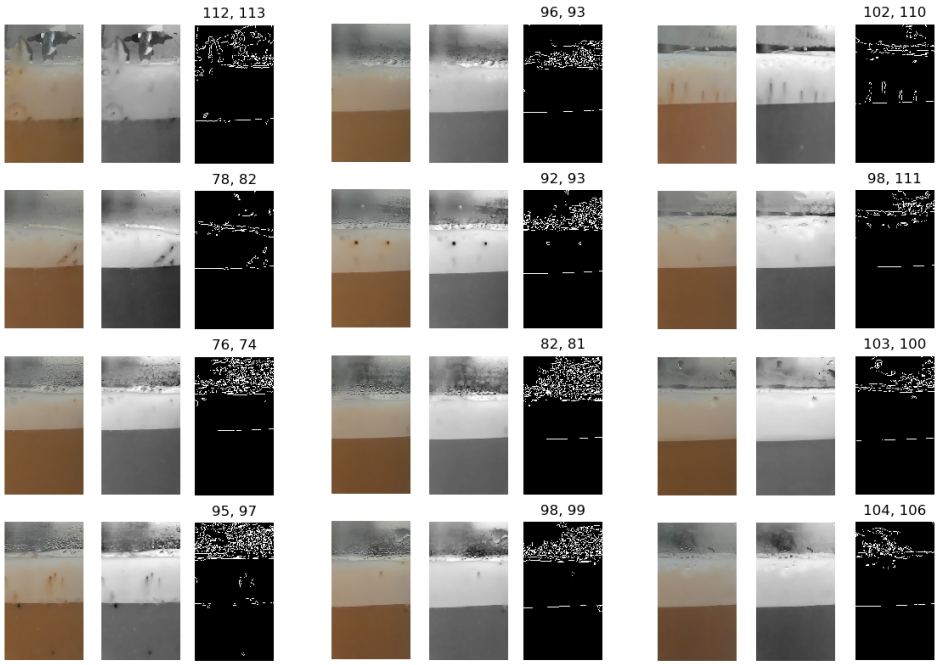


Figure 3.16 Foam height measurement tests to assure versatility, presented in a form of sets <input image, preprocessed image, detected edges> with a mean and mode of measured heights above the last image

The final method, obtained after assuring that it does not return unreasonable values in any of the test images, consisted of six steps. Once the upper part of the side image is cropped, it is converted to grey scale. Consecutive erosion and dilation with kernel of size 3 reduce the noise in the photo. Finally, Canny edge dectection with (10, 120) hysteresis procedure parameters is applied. In each image column, starting the bottom, difference between closest white pixels is calculated. The foam height is a mean of all these differences.

The challenges that appeared during this part of analysis are Canny edge detection parameter's high sensitivity to the light and water condensation on the glass that creates noise in the image. Therefore, it is important to check if the parameters are well tuned before running experiments and that the height measurement starts from the bottom of the photo.

3.3.4 Clump detection

OpenCV offers three gradient filters which could help in clump detection – Laplacian, vertical Sobel and horizontal Sobel filters. Sobel tranforms seemed very promising in the first iteration (Fig. 3.17), however, after further processing, Laplacian proved to be the most suitable indicator of clumps (Fig. 3.18).

To make it easier to analyze, after grayscaling and filtering with Laplacian, pooling is performed. For side image 10x10 window, for 30x30 window is passing through the image and the sum of absolute values is caluclated. The output of such a transformation

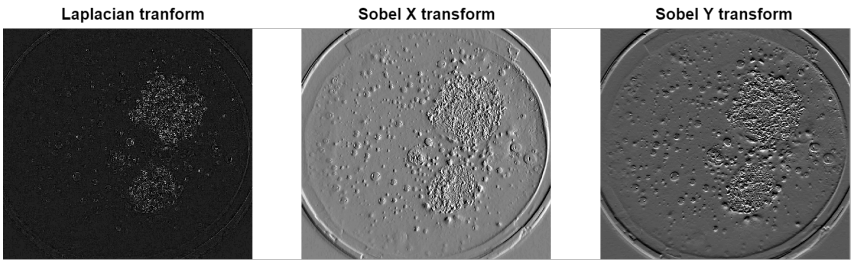


Figure 3.17 First test of gradient transforms

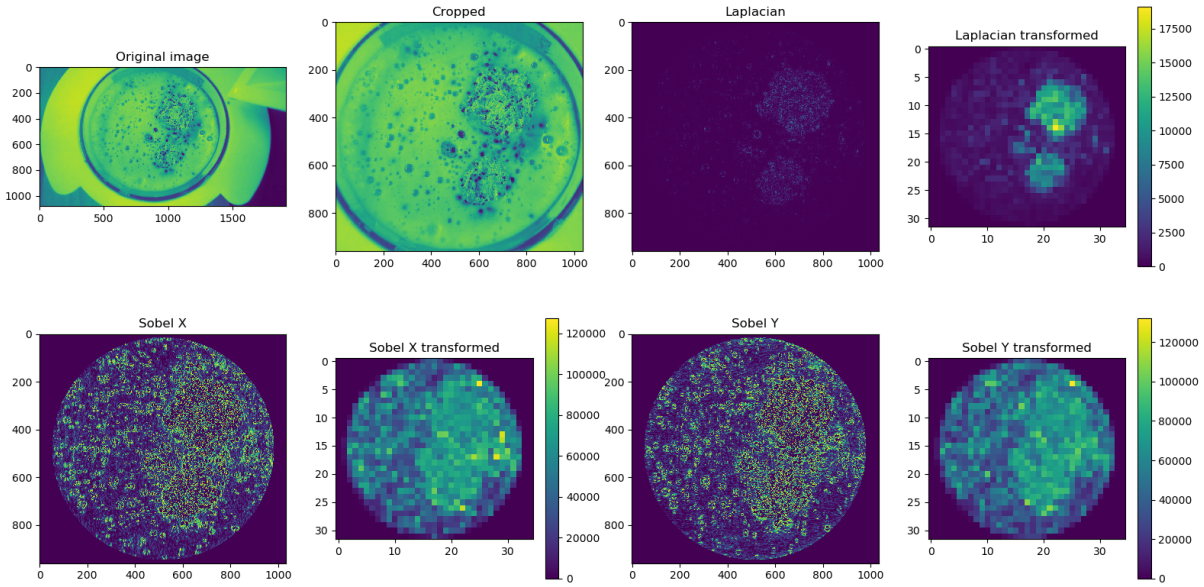


Figure 3.18 Further analysis of clumps and comparison of the methods

is then thresholded – pixels above a trial-and-error-obtained value indicate the presence of a powder clump. The percentage area occupied by the clump is calculacted and returned to the decision making program. The pipelines for clump detection are depicted in Fig. 3.20, 3.19.

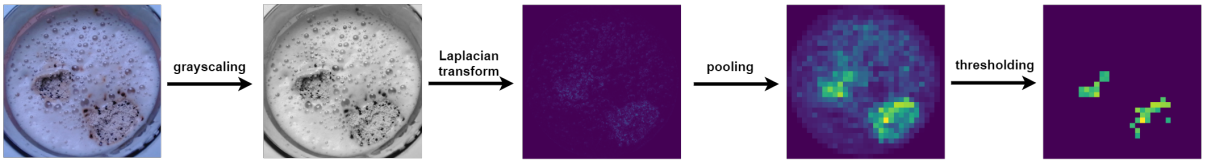


Figure 3.19 Clump in the foam detection pipeline

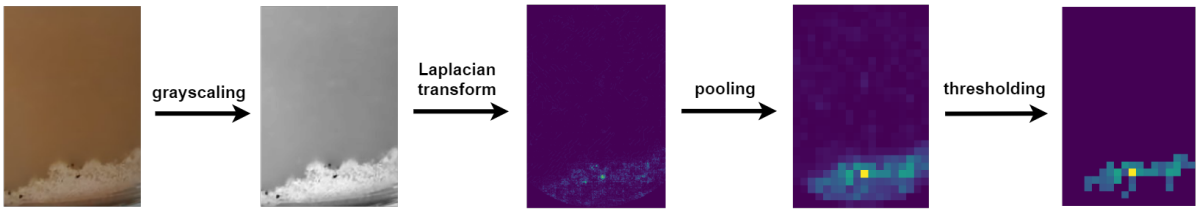


Figure 3.20 Clump in the liquid detection pipeline

3.4 Supplementary software elements

3.4.1 Bash scripts

Several bash scripts and aliases are a part of the Roboccino repository. As the ROS pipeline manages some devices whose parameters need to be passed to configuration files, scripts checking the connection to the robot and the device names of both microcontrollers and cameras were created. Corresponding alias **check_devices** launches these scripts with the example effect in Fig. 3.21. It is much faster and easier to use than checking the device separately with multiple commands.

```
emilia@cappuccino:~/Cappuccino/starting_procedures$ check_devices

Arduino Mega detected. Change device in motor_shield_commander to /dev/ttyACM0
Arduino Uno detected. Change device in servo_commander to /dev/ttyACM1


Side camera detected. Change device in side_camera to /dev/video4
Bubble camera detected. Change device in bubble_camera to /dev/video6


UR5 is connected! :)
```

Figure 3.21 The result of evoking **check_devices** alias in the command line

Running **init_coffee** navigates the user to the ROS workspace with *roboccino* package, builds the packages and sources the local setup bash script. It makes the pipeline simpler to use for people less familiar with ROS. Additionally, instead of calling:

```
$ ros2 launch roboccino system_bringup.launch.py \
                                    mode:={mode_of_operation}
```

every time, aliases in .bashrc file allow for running the command of the same name as the chosen mode of operation. Therefore, running the experiment right after opening the terminal – assuming that in the meantime configuration files are changed accordingly – looks as follows:

```
$ check_devices
$ init_coffee
$ single_experiment_full
```

3.4.2 Microcontroller programs

Programs for both microcontrollers utilized in the system are written in C++ language with the use of Arduino library. Its user-friendly structure and relatively low entry threshold allow for quick implementations without going very low level, which is redundant in the project.

Arduino Uno

Servo library allows for setting the desired position using Pulse Width Modulation. Arduino Uno controls two servos – one from water dispenser on pin 9 and the other one corresponding to the ramp on the pin 10. After receiving four bytes over serial communication, it checks which device is indicated by the first byte. In the case of ramp, the second byte is the desired servo position, so the proper signal is written on the pin. If the device is the water dispenser and if the *on*/*off* values are within correct range, then another condition is checked. If *on* and *off* values are equal, then simply this value is set on the pin, without specified duration time. It is mainly used for switching the water dispenser off. However, if the values are different, then the time interval received in the packet is divided by 2 (as explained in Sect. 3.2.3), *on* value is written on the pin and after the specified time the pin is set as *off*. Finally, if everything was performed properly, received packet is sent back to the computer.

Arduino Mega

Arduino Mega controls DC and stepper motor with the help of Adafruit Motor Shield, therefore the *Adafruit_MotorShield* and *Adafruit_MS_PWMServoDriver* packages are needed for the program. Stirrer is attached to the *M1* pin, while the powder dispenser is operated by *M3*, *M4* pins. After receiving three bytes on serial connection, the selection of the device is checked. If the data packet conserns the powder dispenser, the stepper motor is made to do a number of steps specified by the message with the defined speed. The motor moves backwards with the DOUBLE mode of opration. After performing the task, the motor is released.

In the case stirrer needs to operate, it is run forward with a defined speed. Several delays are called before releasing the motor. It is important to run delays in a loop, because if the time value is too high, passed to the function will be considered as an argument diverging to infinity – the stirrer will not stop functioning.

In both cases, the packet is sent back to the computer as feedback.

Chapter 4

Optimization

4.1 General overview

Finding the input parameters of the system that give the best output is an objective of optimization procedure. The question that was asked in this project was: *Which method to use when optimizing the powdered hot drink preparation?*. Before trying to answer that, inputs and the black-box function's output needed to be determined. There are plenty of factors affecting the powdered coffee preparation – ambient temperature, humidity, stirring motion etc. Various tests were carried out before deciding on the optimization's inputs. Not everything can be easily controlled, so for this setup only three simply changeable parameters with high impact on the bevarage quality are taken into consideration:

- water pouring height h – corresponding to the [0;180] ramp servo positions, however for [90;180] positions height does not change much, so only the range of [0;90] is considered,
- stirring time t – with values between 10 and 60 seconds making the most sense to check,
- stirring speed s – in range 0-100% of DC motor's speed, where reasonable values are between 50% and 100%.

The black-box function's $Q(t, s, h)$ output of the coffee preparation is in the form of a percentage value of area a occupied by the bubbles in the top foam image (Fig. 4.1), being a quality criterion in the case of cappuccinos. The aim of the optimization is to find the minimum of this function. In total, three optimization techniques were explored in this study – Bayesian optimization, Tree-structured Parzen Estimator and grid Bayesian optimization. All of them are run with a corresponding script from the Gitlab repository [Szy21].

The optimization procedure is similar in all cases:

1. A script suggests a set of input parameters to be tested.
2. An operator changes the parameters in *roboccino* package configuration files.

3. Coffee is made and evaluated.
4. The output of the evaluation is put by the operator as feedback to the optimizting program.
5. Another set of input parameters is suggested.

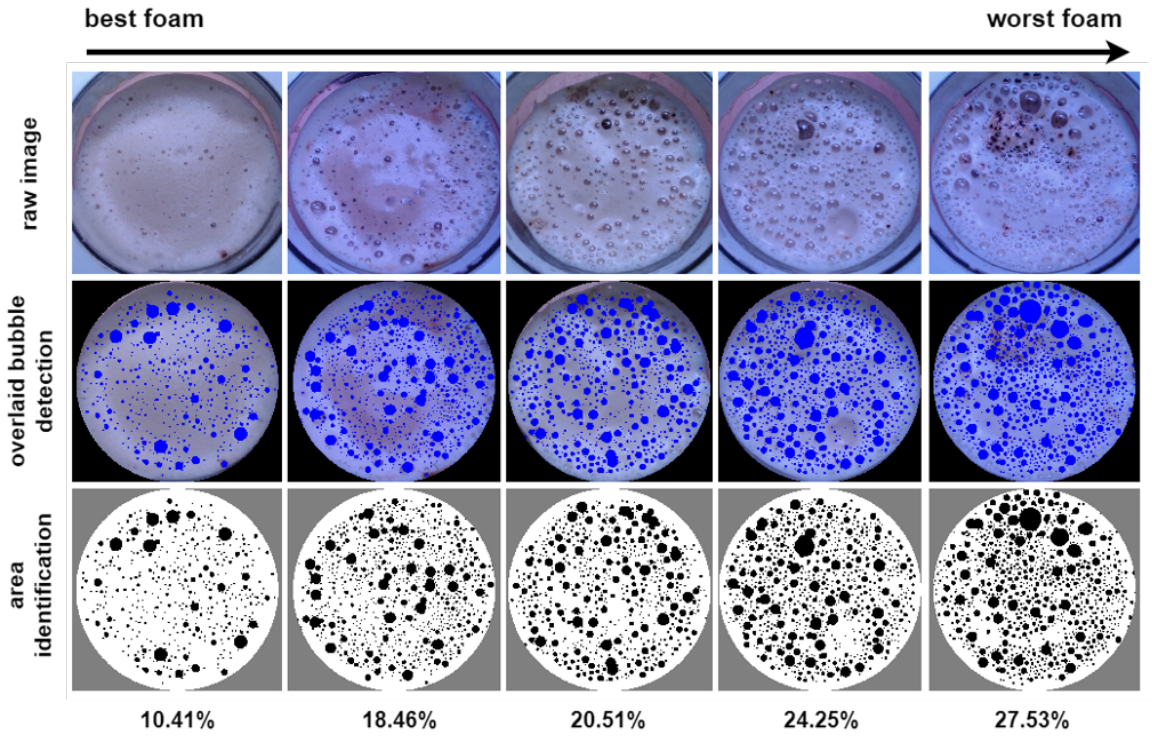


Figure 4.1 Foam quality dependence on the bubble coverage

4.2 Optimization methods

4.2.1 Bayesian optimization

Bayesian optimization's objective is usually to find the global maximum of a black-box function without any assumptions regarding its functional form [Ngu19]. This method operating on the Bayes Theorem treats the examined function as random and puts a prior probability distribution over it. After receiving evaluation data, the prior is accordingly updated and the posterior distribution is formed, which contributes to the creation of an acquisition function pointing out the next query point. The more observations are provided for the optimizer, the more confident the algorithm becomes regarding profitability of certain parameter regions exploration. The Bayesian approach is preferable for cases in which the evaluation of the function is expensive or hard to perform.

The Bayesian optimization has its ready-to-use implementation in Python [Nog14]. There are various methods that determine the prior/posterior distribution over the black-box function and a method using Gaussian process is used in this case. Upper Confidence

Bound as the acquisition function with $\kappa = 8$ and $x_i = 0.1$ result in preference of exploration, whereas the case of $\kappa = 10$ and $x_i = 0.0$ is more in favour of exploitation. As the implemented Bayesian optimization seeks to maximize the output, the problem is in a form of (4.1), because the smaller the bubble area, the better coffee is.

$$f(t, s, h) = 100\% - Q(t, s, h) \tag{4.1}$$

4.2.2 Tree-structured Parzen Estimator optimization

This technique is as well a sequential model-based optimization, however the distributions of observations are modelled using Parzen Etimators [BBBK11]. In contrary to Gaussian-process based methods which would model $P(a \mid (t, s, h))$, in TPE $P((t, s, h) \mid a)$ and $P(a)$ are determined in the computation process. Generative process of parameters is accordingly transformed, distributions of the configuration prior with non-parametric densities are replaced to obtain the probability equal to $P([t, s, h] \mid a)$.

Hyperopt Python library [Hyp13, BYC13] provides tools for easy TPE optimization usage. It has an option to determin the seek for the minimum of the black-box function, therefore the problem can be written as in (4.2).

$$f(t, s, h) = Q(t, s, h) \tag{4.2}$$

4.2.3 Grid Bayesian optimization

The grid Bayesian optimization is a custom method using the same general idea as Bayesian optimization described in Sect. 4.2.1, however one simplification is added. Only predictions for water pouring height and stirring speed are made. For suggested pairs of parameters, stirring is performed for $T = \{15, 30, 45, 60\}$ seconds. The best result out of these four cases is fed into the optimizer and the next set of parameters is returned. Parameter number reduction was hoped to speed up the process of convergence in parameter space exploration. The examined function can be defined as in (4.3).

$$f(s, h) = 100\% - Q(T, s, h) \tag{4.3}$$

4.3 Results

In total, about two hundred coffees have been prepared while performing experiments. All the results concerning prepared coffees are plotted on the Fig. 4.2. It is evident that for the stirring speed lower than 60% and mixing time below 30 seconds the coffee quality distinctly drops. The highest condensation of low bubble area outputs is in the region of stirring for 30-50 seconds and with speed above 80%. Considering water pouring height, it is hard to indicate a trend – it may be due to the need of a greater number of experiments, narrow height range or low bubble area dependence on this parameter.

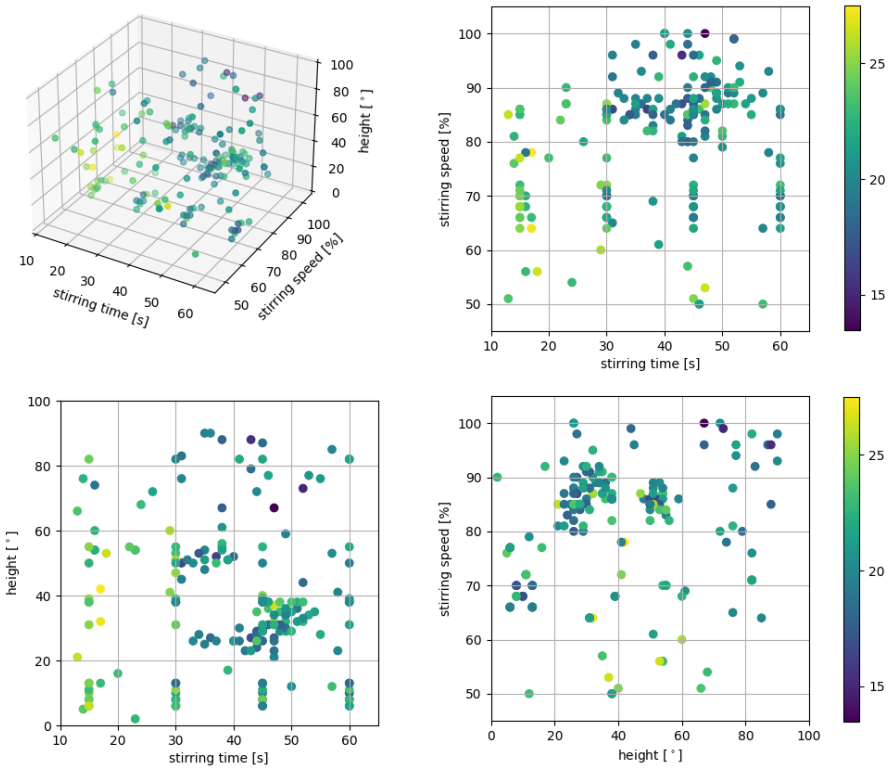


Figure 4.2 Experiments results in relation to area covered by bubbles (in %)

To check how stochastic the preparation process is, variance of coffees prepared with the random input parameters has been tested (Fig. 4.3) and proved to be of acceptable magnitude. However, it later became apparent that some of the results can have variance of much greater value, which final comparison depicts in Fig. 4.5.

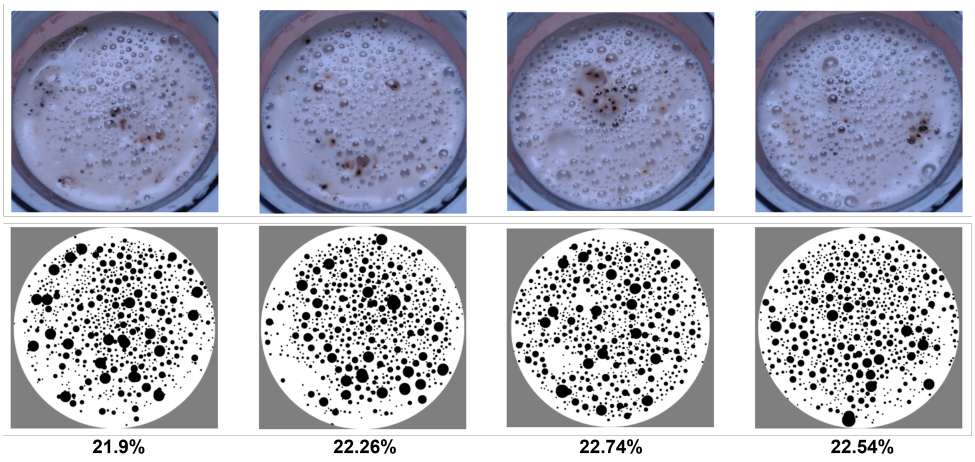


Figure 4.3 Foam variance

Four series of experiments have been performed – two for Bayesian optimization, one for TPE and one for grid Bayesian, all depicted in Fig. 4.4. In the case of Bayesian optimization, one additional series with $\kappa = 10$ and $x_i = 0.0$ was carried out to check the behaviour of the optimizer.

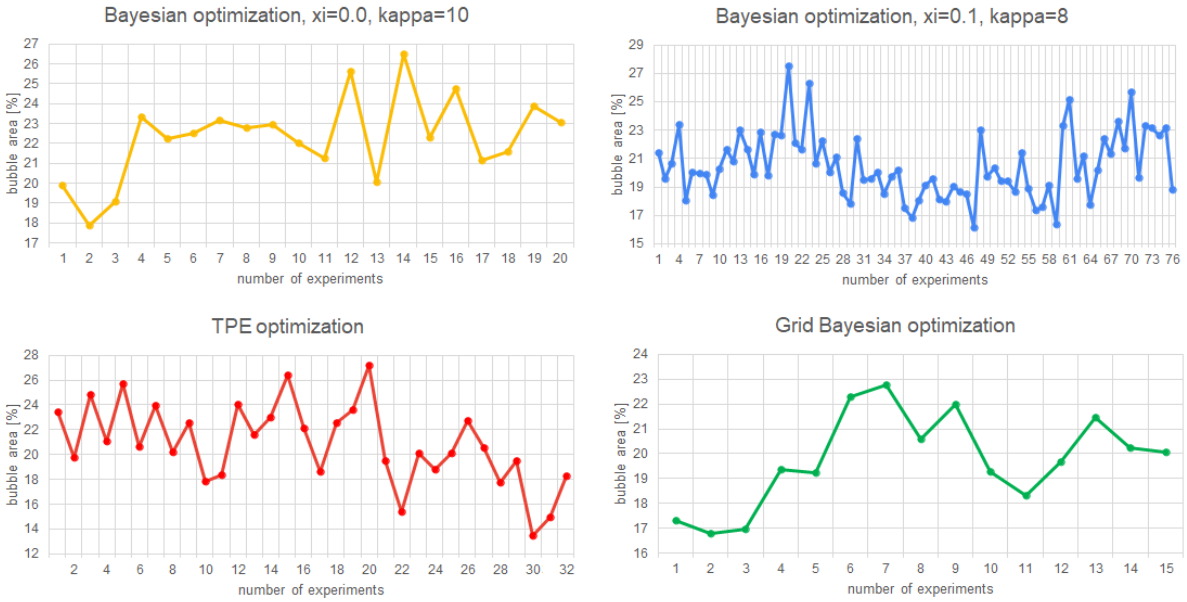


Figure 4.4 Bubble area in the coffees throughout the experiments – two Bayesian optimizations, TPE optimization and grid Bayesian optimization

The sets of parameters resulting in the best coffee found by each of the method, along with the human-made coffee have been repeated a couple of times and compared (Fig. 4.5). Additionally, a random optimization process was performed – the best result was added to the comparison.

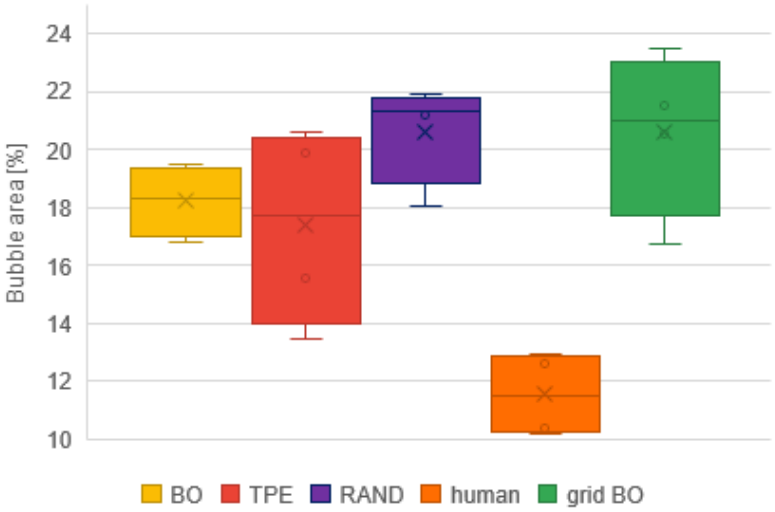


Figure 4.5 Comparison of the best results obtained by the each parameter search method

It is visible at a glance that a human-prepared coffee is much better than one made with a robotic setup with open loop control. Although TPE provided some cases with very low bubble area, a very high variance came with it. Taking everything into consideration, it is hard to choose the best optimization method at this point as none of them fully converged to the best solution. Further experiments would be needed to verify the correctness of the techniques' outputs.

The effects of adding a closed-loop control can be observed in Fig. 4.6. A trend of coffee quality increase (in terms of both clump elimination and bubble coverage decrease) can be seen in consecutive iterations (with some exceptions).

No search in optimal determination of stirring speed and time in consecutive loop runs was performed as it is a much more complex task – it is impossible to have the same optimization's starting point after initial mixing, which is proven by coffee high variance as in Fig. 4.3. However, it can be seen that multiphase mixing on a different height depending on the clump detection positively affects cappuccino's quality.

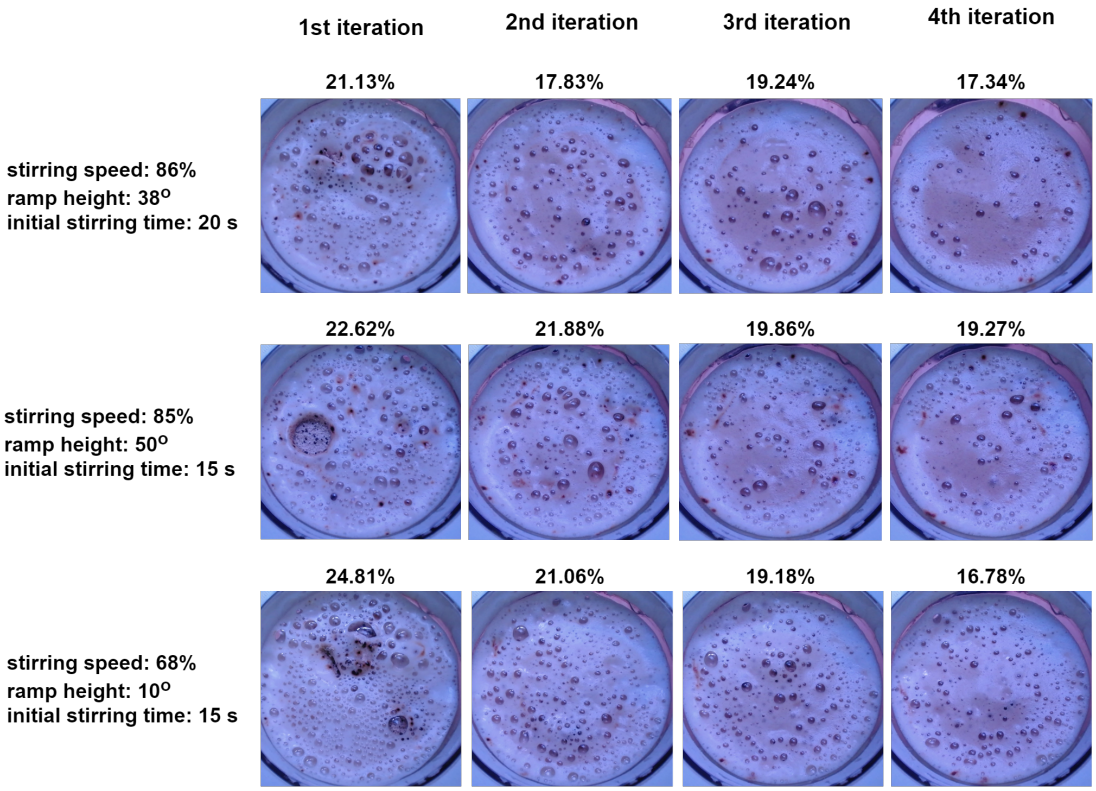


Figure 4.6 Examples of closed loop control iterations

Chapter 5

Conclusions

The goal of this thesis was to develop a system for robotized coffee preparation. Achieving the goal required to design, 3D print and compose a hardware system, program it, apply computer vision to closed loop control and carry out experiments with search for the optimal input parameters. Cappuccino was the case study analyzed in the project, with its powdered substitute eventually used to prepare hot drinks. The initial problem investigation defined the tasks of the robotic setup – it had to allow for picking up a cup, collecting the powder, stirring while pouring the water and evaluation of the cappuccino quality. After subject overview, the hardware elements were designed and composed from commercially available products (such as UR5 robot, microcontrollers, cameras) and 3D printed parts. The robot with its custom end effector, water and powder dispenser, stirrer, cup rims, positioners and water ramp made it possible for carrying out the actions leading to objective completion. The next step was to implement the software system. Apart from programming the work of the robotic arm and other programmable hardware elements, the software had to contain image analysis for coffees assessment and closed-loop control for multiphase improvement. It was all wrapped in ROS2 Python architecture, with some auxiliary bash scripts and microcontrollers' programs in C++. The final stage of the project consisted of experiments. Optimization methods for black-box functions were explored, series of tests planned and executed with applied optimization techniques. The results of the optimal parameters search were analyzed and conclusions on the optimal process parameters to obtain the best coffee were presented.

Created system met the requirements defined in the study objective – cappuccino is prepared in an automated and parameterized way, which is helpful with carrying out multiple experiments. If the state of the drink is classified as not satisfactory, the robot performs additional mixing in accordance to the output of computer vision pipeline. As microfoam is the desired output, milk foam assessment is based on the percentage of bubble area in the image taken from the above the cup. High stochasticity – visible in redoing coffees with the same input parameters – negatively affects the exact repeatability characteristics and disturbes the optimization procedure. Nevertheless, the computer vision feedback was proven to be a valuable component of the system. Blob detection and adaptive thresholding were the main tools determining it. However, clumped powder in both top and side images also needed to be detected and eliminated in closed loop

control, so Laplacian transform with pooling were used in the corresponding part of the software. Initially foam layer height was also supposed to be taken into consideration, but it turned out to be not necessary – its values did not differ much between high and low quality bevarages. OpenCV library was easily imported and utilized in the custom ROS package, making it a suitable tool for data analysis. Apart from stand-alone coffee evaluation, image processing made it possible to improve the bevarage by foam bubbles reduction and powder clumps elimination.

The final step was to perform an optimization in search for input parameters resulting in the best cappuccino. Water pouring height, mixing speed and stirring time affected the formation of the milk foam, therefore they were considered as input parameters for Bayesian, Tree-Structured Parzen Estimator and grid Bayesian optimization methods. Although it is not yet possible to point out the best optimization method, a delicate trend regarding the good input parameters can be seen. Best coffees were made with stirring time around 30-50 seconds and mixing speed near 80-100%. It is hard to observe a tendency in water pouring height - the reason for that may be a small range of available heights (maximum difference was about 2 cm).

Further experiments would be needed to clarify trends for input-output relation. However, apart from tests, some improvements could be added to extend the project. Wider range of water pouring height, total setup separation from external factors (such as varying temperture or humidity in the lab) and exploration in higher DC motor's input voltage might positively affect the system's operation. A task, that may be interesting to investigate, but also hard to put into action, would be optimization for closed-loop control. It would require low variance in coffee preparation, so development of system elements in terms of exact repeatability would be necessary to undertake. Different approaches towards image processing could be tested and compared - perhaps a better solution would appear.

Taking all the above into consideration, a major conclusion emerged – it is highly challenging to implement actions that a human performs without much focus or dedication. Although many simplifications (such as substituting classic cappuccino with its powdered version) were applied, it still took a lot of time to construct and program the setup, which is not even close in efficiency to human vision analysis and continuous closed-loop control. Plenty of scientists encounter this barrier and hopefully progressive research in robotics will soon propose a solution bringing us closer to achieving high effectiveness.

References

[BBBK11] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011.

[BBR11] Mario Bollini, Jennifer Barry, and Daniela Rus. Bakebot: Baking cookies with the pr2. In *The PR2 workshop: results, challenges and lessons learned in advancing robots with a common platform, IROS*, 2011.

[BKK+11] Michael Beetz, Ulrich Klank, Ingo Kresse, Alexis Maldonado, Lorenz Mösenlech-ner, Dejan Pangercic, Thomas Rühr, and Moritz Tenorth. Robotic roommates making pancakes. In *2011 11th IEEE-RAS International Conference on Humanoid Robots*, pages 529–536, October 2011. ISSN: 2164-0572.

[BMS09] E.M. Becker, L.S. Madsen, and L.H. Skibsted. Storage stability of cappuccino powder. *Milchwissenschaft*, 64:413–417, 01 2009.

[BYC13] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 115–123, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[CBK+21] Apeksha Chipade, Pallavi Bhagyawant, Pratiksha Khade, Rajshri C. Mahajan, and Vibha Vyas. Computer vision techniques for crowd density and motion direction analysis. In *2021 6th International Conference for Convergence in Technology (I2CT)*, pages 1–4, 2021.

[Cib] Cibo360. Cappuccino – Italian recipe. https://www.cibo360.it/alimentazione/cibi/caffe/cappuccino.htm. Accessed: 2021-10-30.

[CL08] Yaohuan Cui and Chang Woo Lee. Vision-based human motion analysis for event recognition. In *2008 Second International Symposium on Intelligent Information Technology Application*, volume 2, pages 263–267, 2008.

[con20] Wikipedia contributors. Microfoam — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Microfoam, 2020. Accessed: 01-July-2021.

[DKD15] J. Dorner, Š. Kozák, and F. Dietze. Object recognition by effective methods and means of computer vision. In *2015 20th International Conference on Process Control (PC)*, pages 198–202, 2015.

[GL10] Kristen Gauman and Bastian Leibe. *Visual Object Recognition*. Morgan & Claypool Publishers, 2010.

[Hof00] W Hoffmann. Characterization of cappuccino powders. *Kieler Milchwirtschaftliche Forschungsberichte*, 52:165–174, 01 2000.

[Hua96] Thomas S. Huang. Computer vision: Evolution and promise. 1996. Presented at 5th International Conference on High Technology.

[Hyp13] Hyperopt: Distributed hyperparameter optimization python library. https://github.com/hyperopt/hyperopt, 2013. Accessed: 2021-08-30.

[IKK17] Jamshed Iqbal, Zeashan Hameed Khan, and Azfar Khalid. Prospects of robotics in food industry. *Food Science and Technology*, 37:159–165, 2017.

[JHTI20] Kai Junge, Josie Hughes, Thomas George Thuruthel, and Fumiya Iida. Improving robotic cooking using batch bayesian optimization. *IEEE Robotics and Automation Letters*, 5(2):760–765, 2020.

[Jur06] AC Juriaanse. Challenges ahead for food science. *International journal of dairy technology*, 59(2):55–57, 2006.

[KKI18] Zeashan Hameed Khan, Azfar Khalid, and Jamshed Iqbal. Towards realizing robotic potential in future intelligent food manufacturing systems. *Innovative food science & emerging technologies*, 48:11–24, 2018.

[LC20] Rich Lee and Ing-Yi Chen. The time complexity analysis of neural network model configurations. In *2020 International Conference on Mathematics and Computers in Science and Engineering (MACISE)*, pages 178–183, 2020.

[MHB12] Slavomir Matuska, Robert Hudec, and Miroslav Benco. The comparison of cpu time consumption for image processing algorithm in matlab and opencv. In *2012 ELEKTRO*, pages 75–78, 2012.

[Ngu19] Vu Nguyen. Bayesian optimization for accelerating hyper-parameter tuning. In *2019 IEEE Second International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*, pages 302–305, 2019.

[Nog14] Fernando Nogueira. Bayesian Optimization: Open source constrained global optimization tool for Python. https://github.com/fmfn/BayesianOptimization, 2014. Accessed: 2021-08-30.

[Rob] Moley Robotics. Moley world's first robotic kitchen. https://www.moley.com. Accessed: 2021-10-30.

[Ros88] A. Rosenfeld. Computer vision: basic principles. *Proceedings of the IEEE*, 76(8):863–868, 1988.

[Sch07] Harald Schuchmann. Product design for coffee based beverages. *Product Design & Engineering*, 2, 2007.

[Sci] Blob detection methods in scikit-image python library. https://scikit-image.org/docs/stable/auto_examples/features_detection/plot_blob.html. Accessed: 2021-10-30.

[SHIH21] Grzegorz Sochacki, Josephine Hughes, Fumiya Iida, and Simon Hauser. Closed-loop robotic cooking of scrambled eggs with a salinity-based 'taste'sensor. *International Conference on Intelligent Robots and Systems*, 2021.

[SIH21] Grzegorz Sochacki, Fumiya Iida, and Josephine Hughes. Compliant sensorized testing device to provide a model-based estimation of the cooking time of vegetables. *16th International Conference On Intelligent Autonomous System*, 2021.

[SMCC+19] Luca Scimeca, Perla Maiolino, Daniel Cardin-Catalan, Angel P. del Pobil, Antonio Morales, and Fumiya Iida. Non-destructive robotic assessment of mango ripeness via multi-point soft haptics. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 1821–1826, 2019.

[SRLS16] Aykut Satici, Fabio Ruggiero, Vincenzo Lippiello, and Bruno Siciliano. A coordinate-free framework for robotic pizza tossing and catching. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3932–3939, 05 2016.

[SSM+20] Deotale Shweta, Dutta Sayantani, JA Moses, VM Balasubramaniam, and C Anandharamakrishnan. Foaming characteristics of beverages and its relevance to food processing. *Food Engineering Reviews*, 12(2):229–250, 2020.

[Szy21] Emilia Szymańska. Roboccino gitlab repositories: cappuccino preparation robotic system. https://gitlab.com/roboccino, 2021.

[TM09] Hely Tuorila and Erminio Monteleone. Sensory food science in the changing society: Opportunities, needs, and challenges. *Trends in Food Science & Technology*, 20(2):54–62, 2009.

[XWD13] Yuan-nan Xu, Jing Wang, and Yan-bing Dong. Mixed norm-based image restoration using neural network. In *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, pages 1957–1961, 2013.

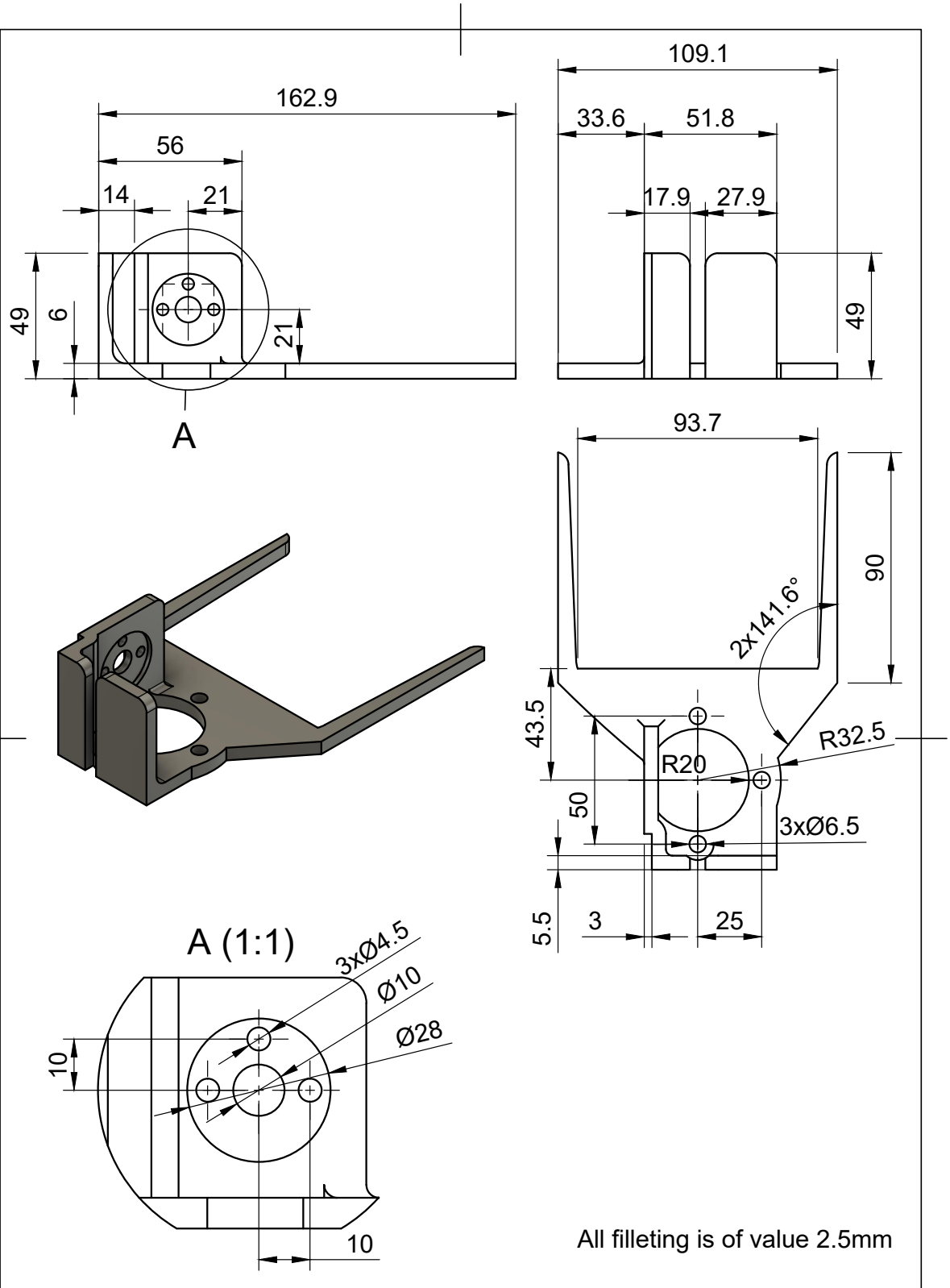
[Zha10] Bo Zhang. Computer vision vs. human vision. In *9th IEEE International Conference on Cognitive Informatics (ICCI'10)*, page 3, 2010.

[Zou19] Xinrui Zou. A review of object detection techniques. In *2019 International Conference on Smart Grid and Electrical Automation (ICSGEA)*, pages 251–254, 2019.

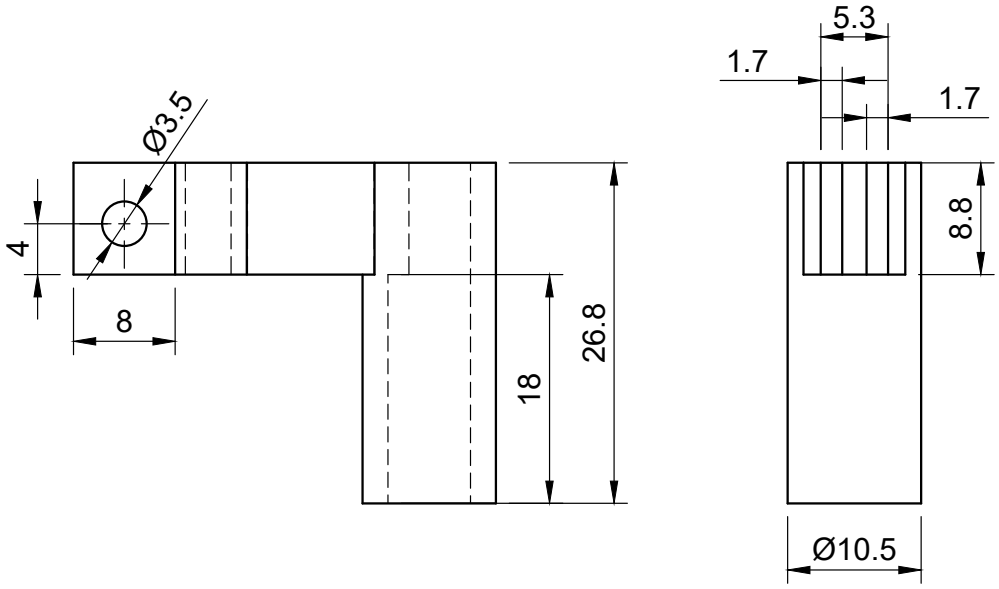
Appendix A

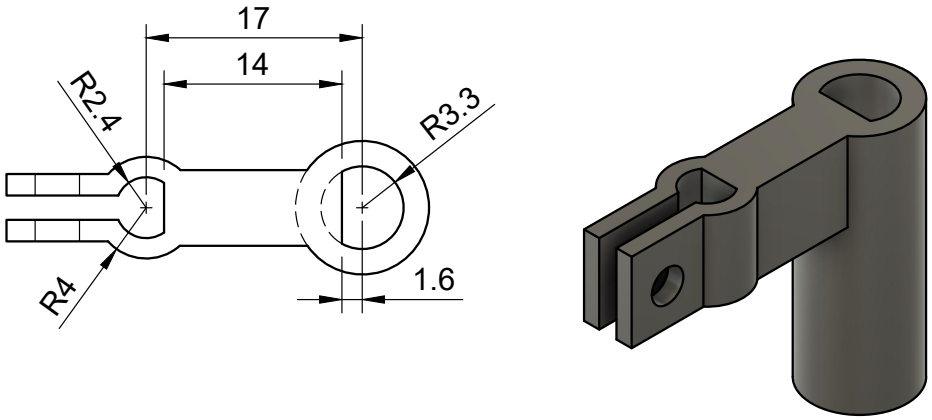
Technical drawings of custom 3D elements

In this appendix, technical drawings of all custom 3D designed elements are attached.

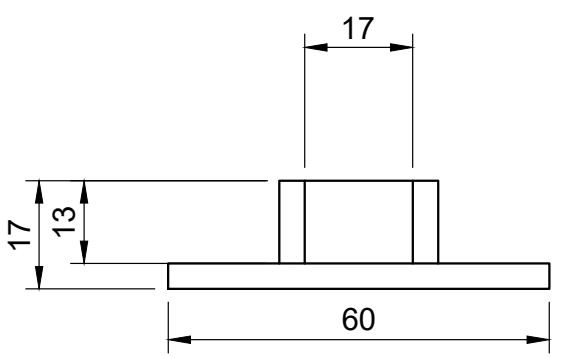


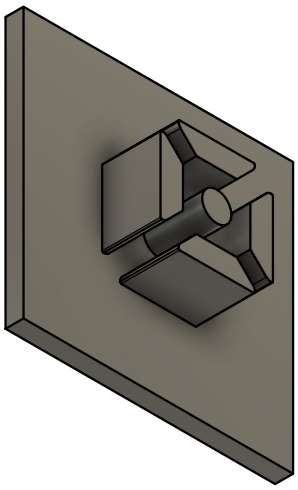
Created by	Emilia Szymańska	Project name	End effector
Date	30.08.2021	Element name	Gripper base
Material	PLA	Scale	1:2
Units	mm	Drawing No	1

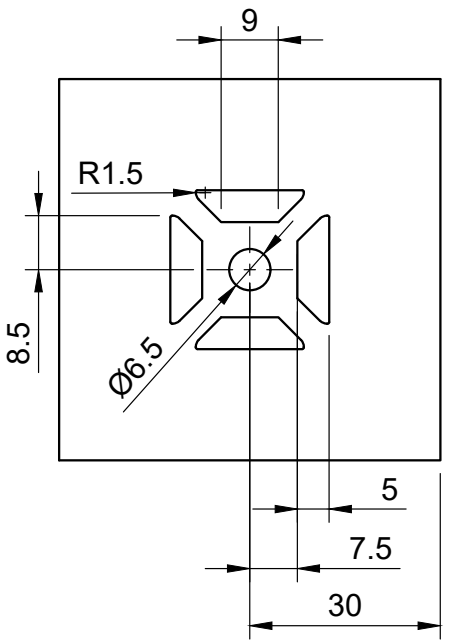


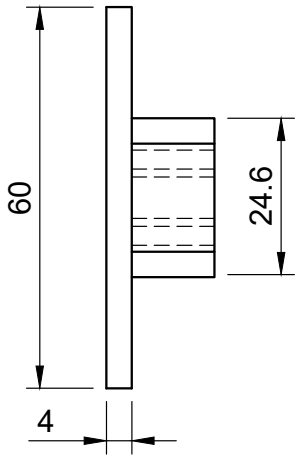


Created by	Emilia Szymańska	Project name	End effector
Date	30.08.2021	Element name	Stirrer holder
Material	PLA	Scale	2:1
Units	mm	Drawing No	2



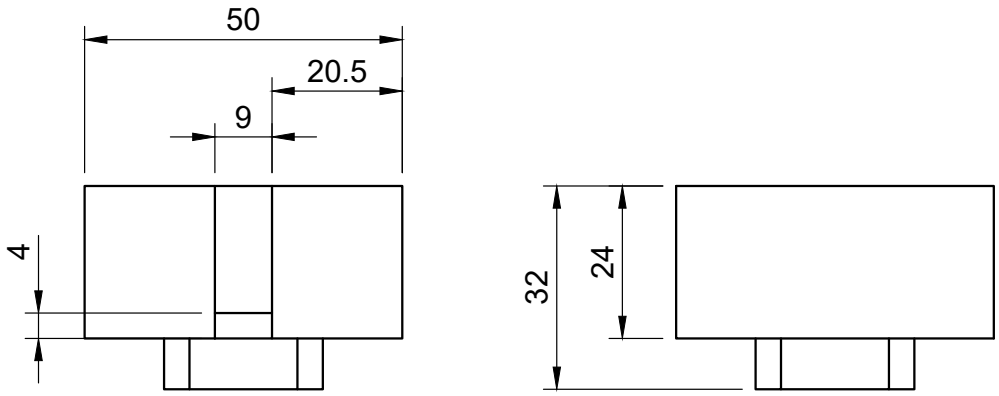


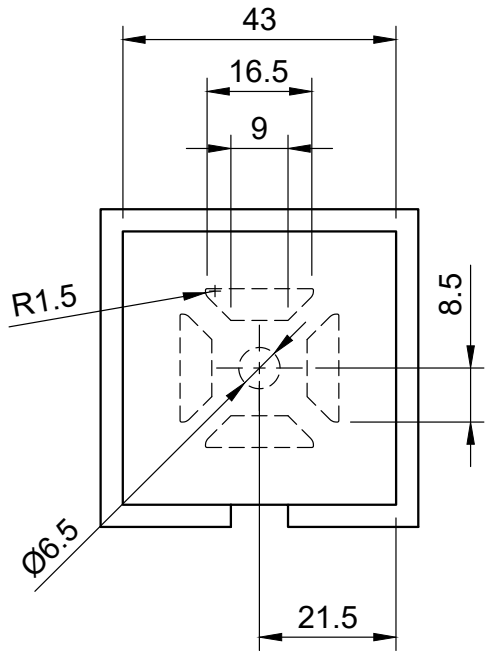


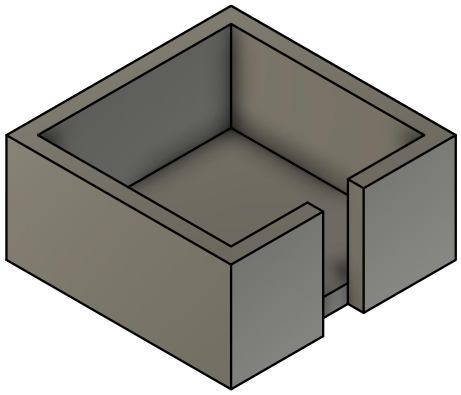


Trapezoidal convex elements are of the same dimensions

Created by	Emilia Szymańska	Project name	Powder dispenser
Date	30.08.2021	Element name	Dispenser base
Material	PLA	Scale	1:1
Units	mm	Drawing No	3

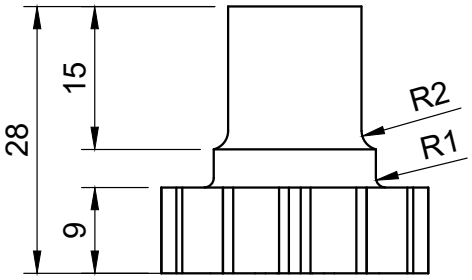


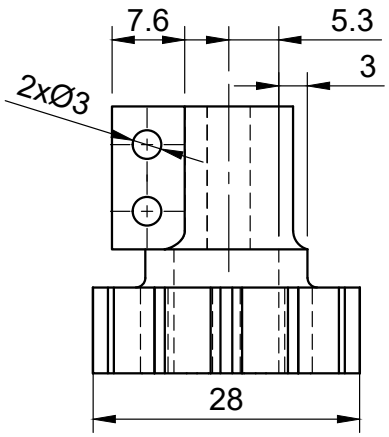


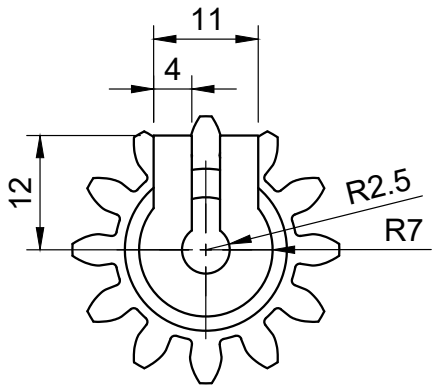


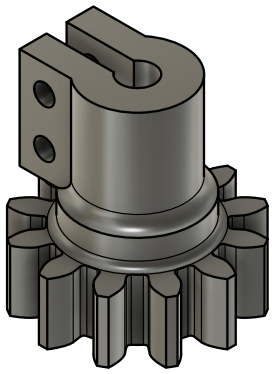
Trapezoidal convex elements are of the same dimensions

Created by	Emilia Szymańska	Project name	Powder dispenser
Date	30.08.2021	Element name	Stepper motor holder
Material	PLA	Scale	1:1
Units	mm	Drawing No	4





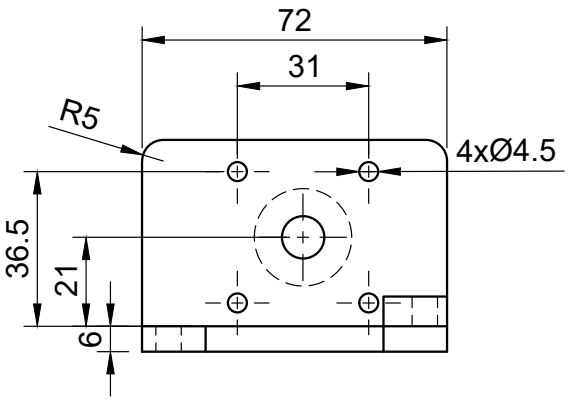


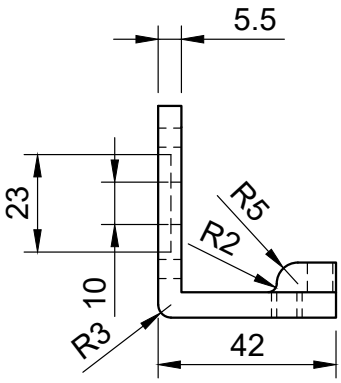


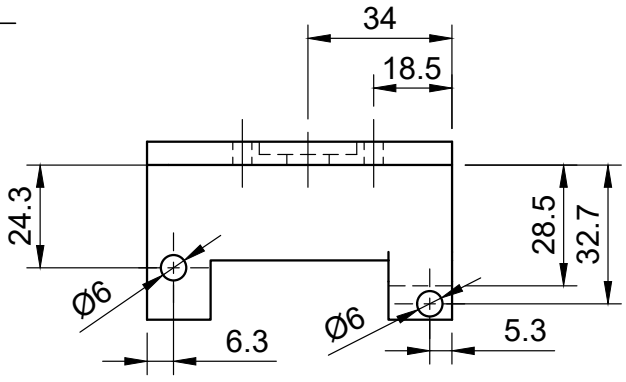
Gear parameters:

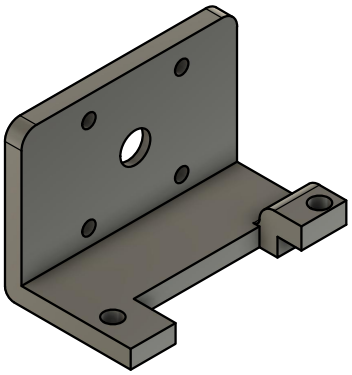
- pitch diameter 24 mm
- pressure angle 20 deg
- root fillet radius 5 mm
- gear thickness 9 mm
- hole diameter 11 mm
- teeth number 12

Created by	Emilia Szymańska	Project name	Powder dispenser
Date	30.08.2021	Element name	Gear
Material	PLA	Scale	1.5:1
Units	mm	Drawing No	5

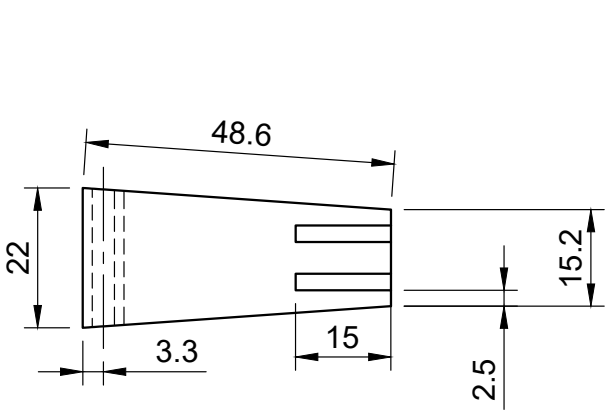


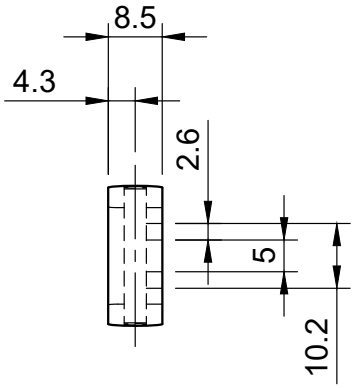


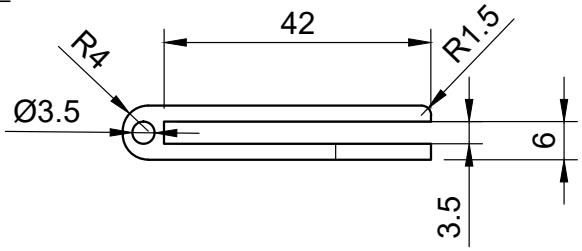


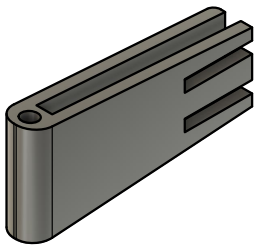


Created by	Emilia Szymańska	Project name	Powder dispenser
Date	30.08.2021	Element name	Stepper motor mount
Material	PLA	Scale	1:1.5
Units	mm	Drawing No	6

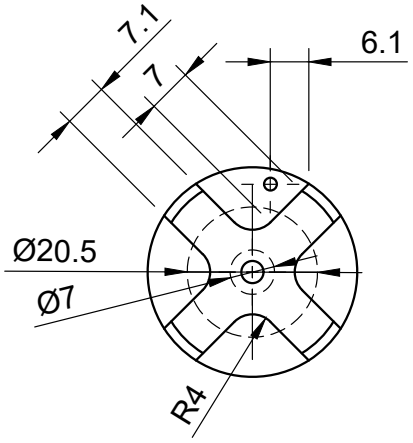


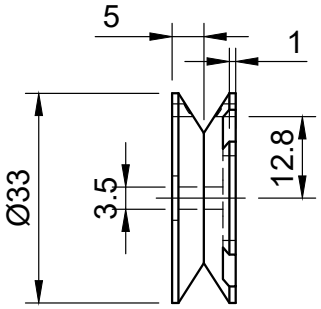


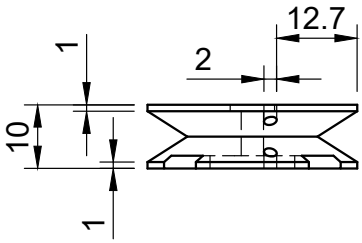


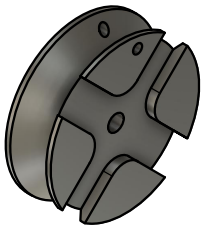


Created by	Emilia Szymańska	Project name	Water dispenser
Date	30.08.2021	Element name	Lever overlay
Material	PLA	Scale	1:1
Units	mm	Drawing No	7

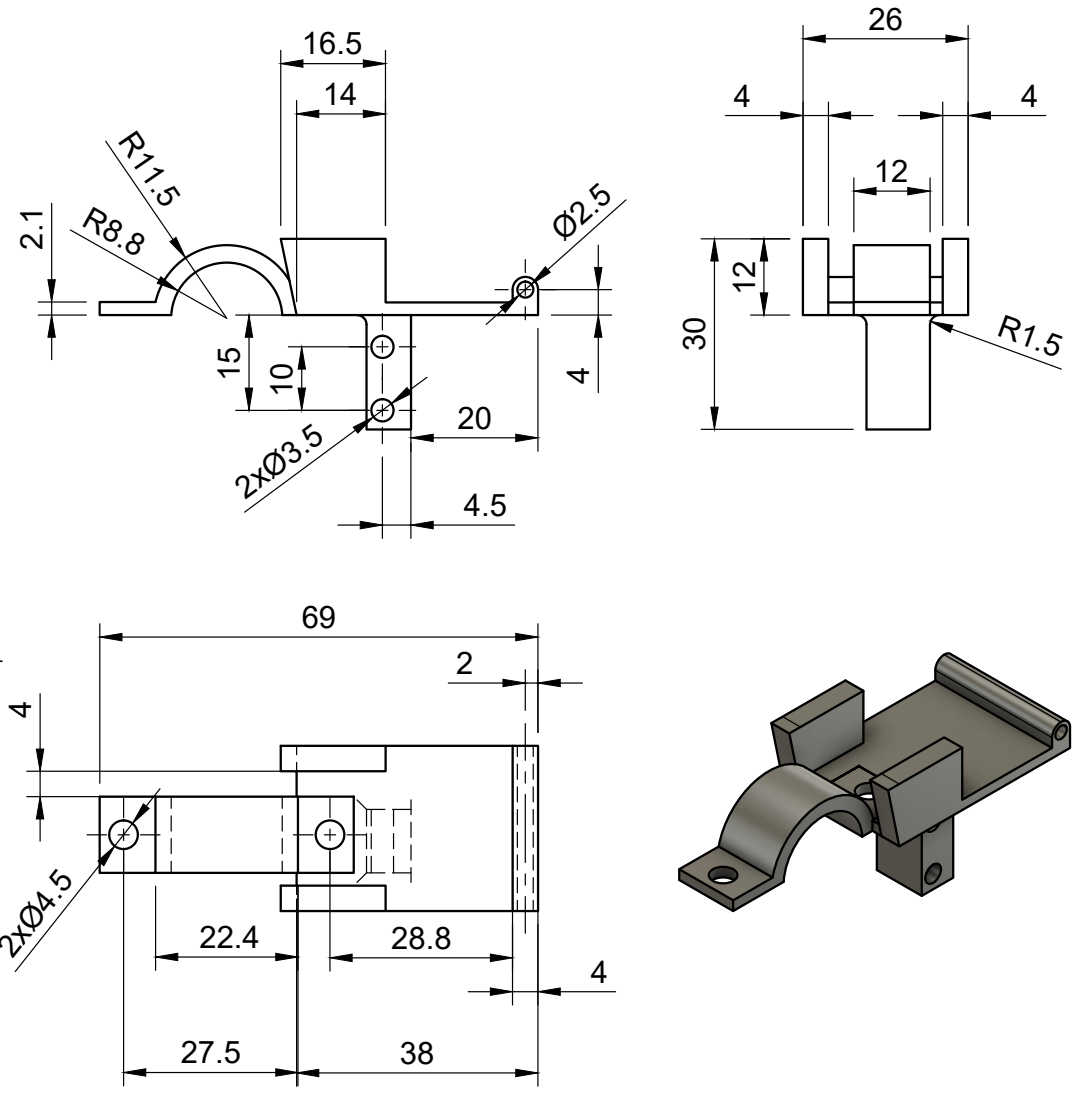




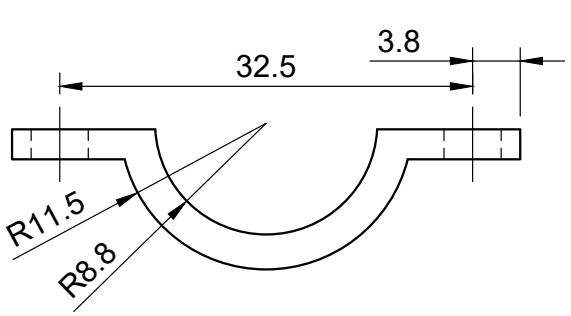


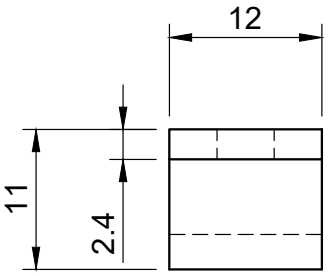


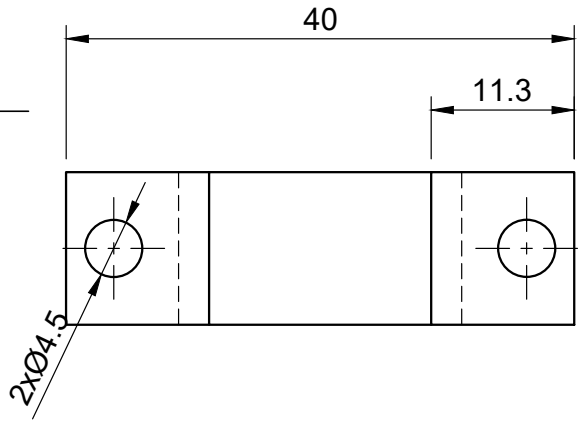
Created by	Emilia Szymańska	Project name	Water dispenser
Date	30.08.2021	Element name	Pulley
Material	PLA	Scale	1:1
Units	mm	Drawing No	8

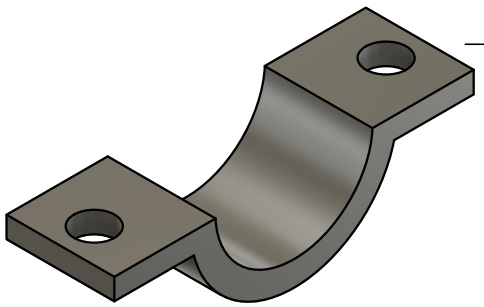


Created by	Emilia Szymańska	Project name	Water dispenser
Date	30.08.2021	Element name	Servo mount
Material	PLA	Scale	1:1
Units	mm	Drawing No	9

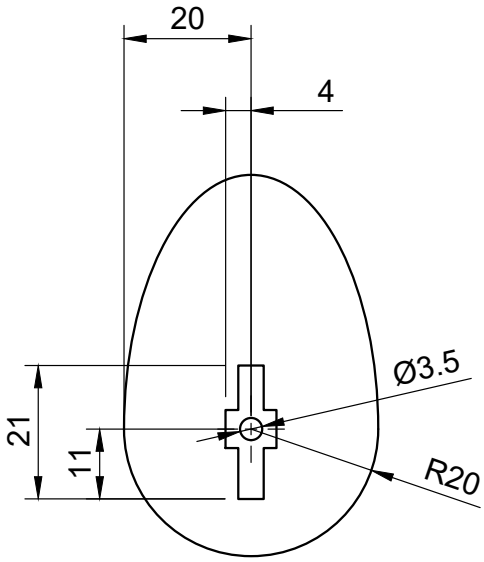


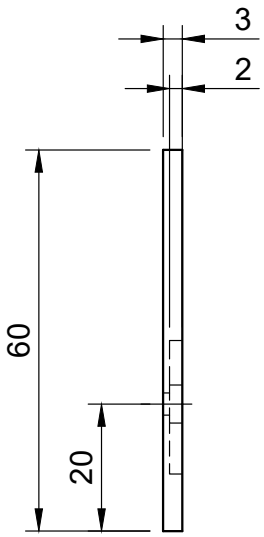


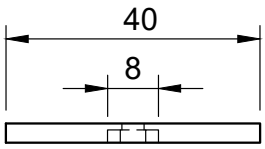


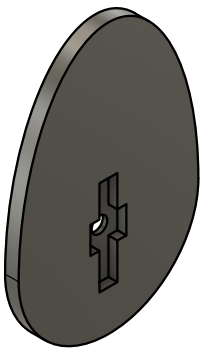


Created by	Emilia Szymańska	Project name	Water dispenser
Date	30.08.2021	Element name	Lower clamp band
Material	PLA	Scale	2:1
Units	mm	Drawing No	10

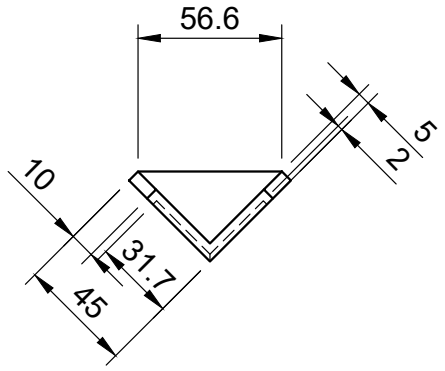


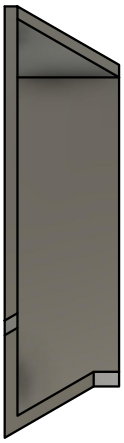


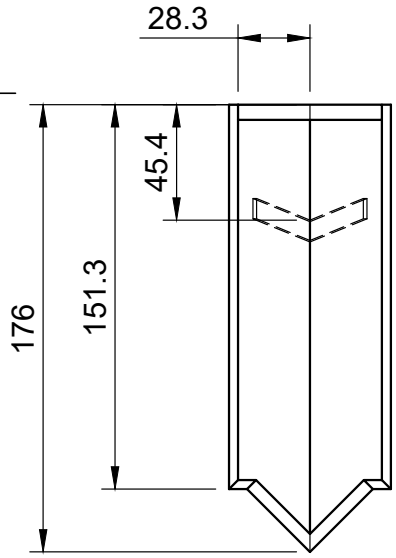


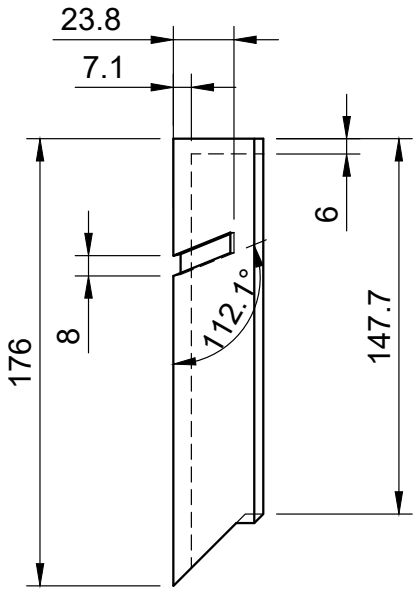


Created by	Emilia Szymańska	Project name	Ramp
Date	30.08.2021	Element name	Cam
Material	PLA	Scale	1:1
Units	mm	Drawing No	11

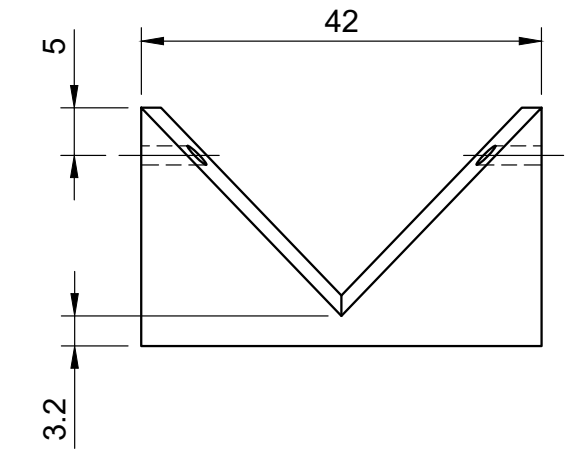


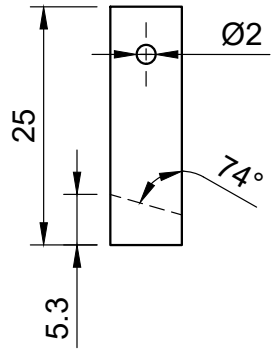


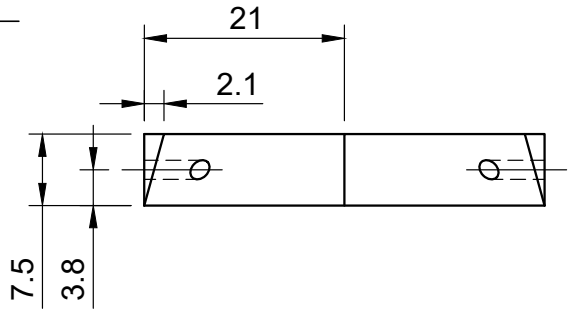


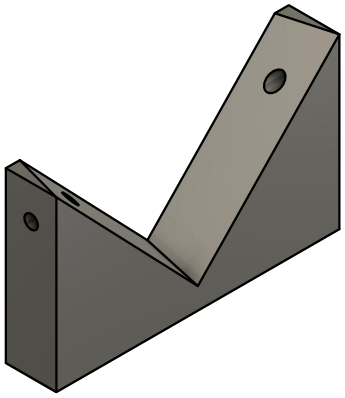


Created by	Emilia Szymańska	Project name	Ramp
Date	30.08.2021	Element name	Water channel
Material	PLA	Scale	1:2.5
Units	mm	Drawing No	12

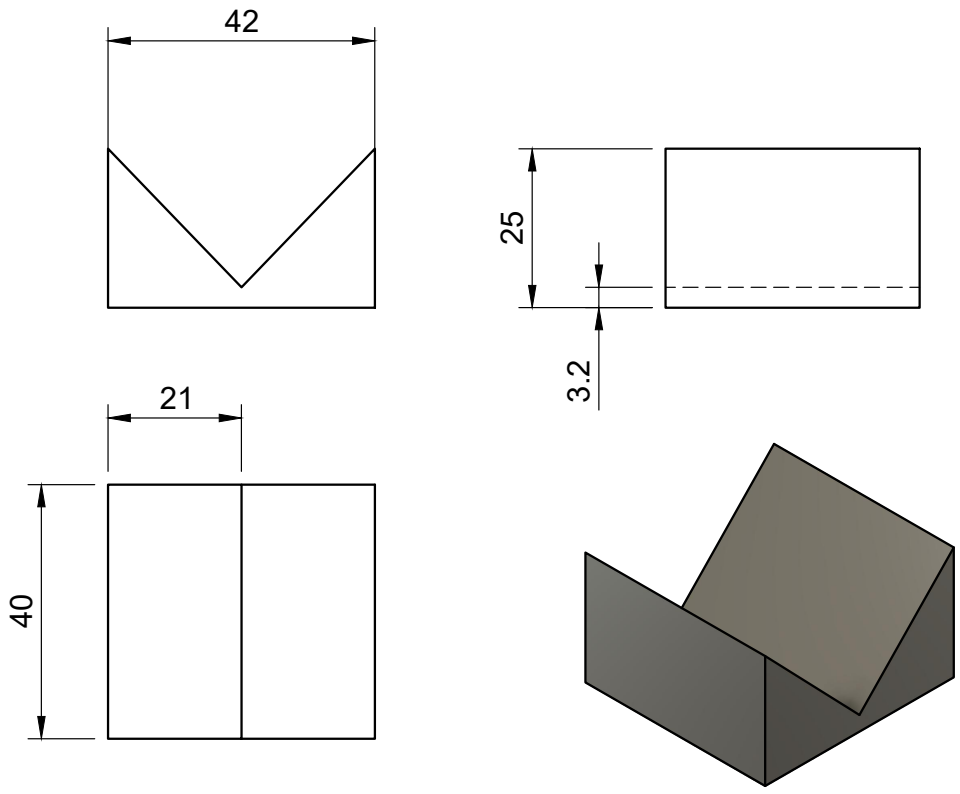




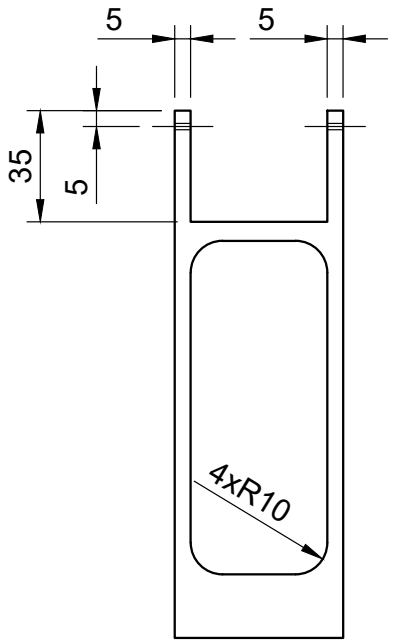


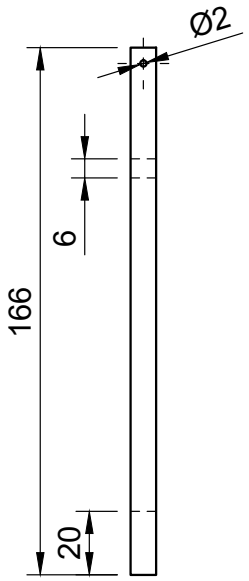


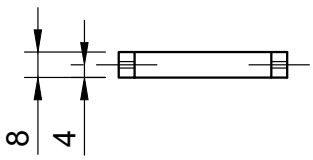
Created by	Emilia Szymańska	Project name	Ramp
Date	30.08.2021	Element name	Upper distancer
Material	PLA	Scale	1.5:1
Units	mm	Drawing No	13

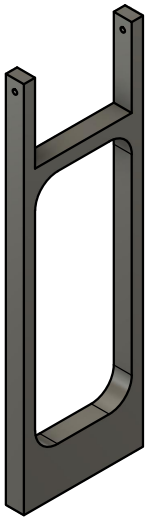


Created by	Emilia Szymańska	Project name	Ramp
Date	30.08.2021	Element name	Back distancer
Material	PLA	Scale	1:1
Units	mm	Drawing No	14

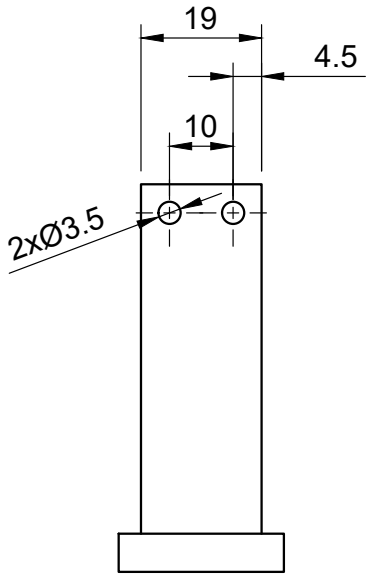


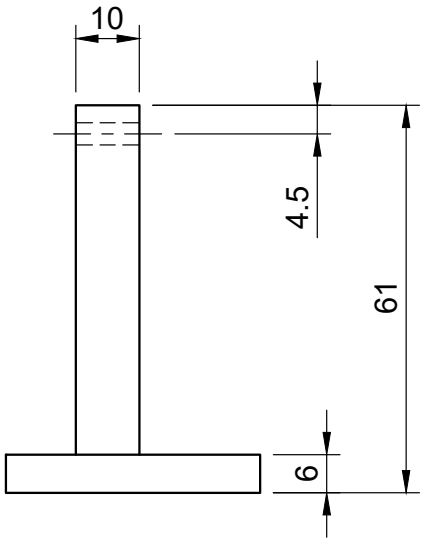


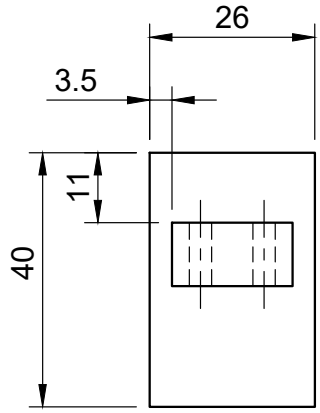


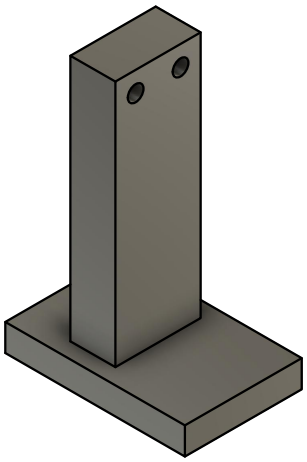


Created by	Emilia Szymańska	Project name	Ramp
Date	30.08.2021	Element name	Column
Material	PLA	Scale	1:2
Units	mm	Drawing No	15

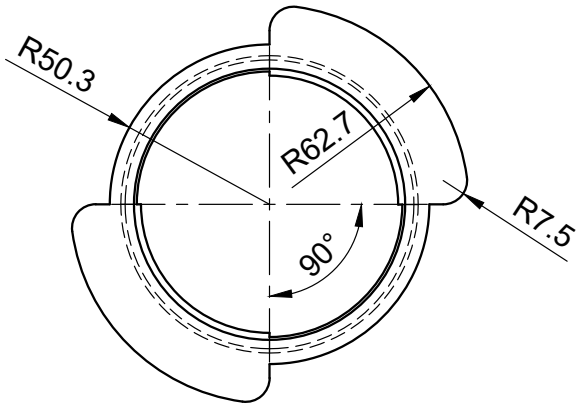


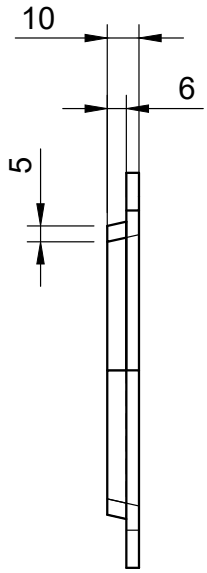


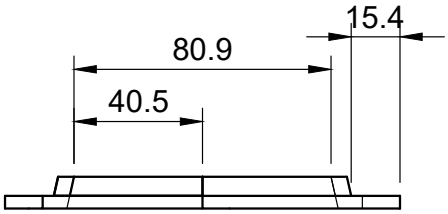




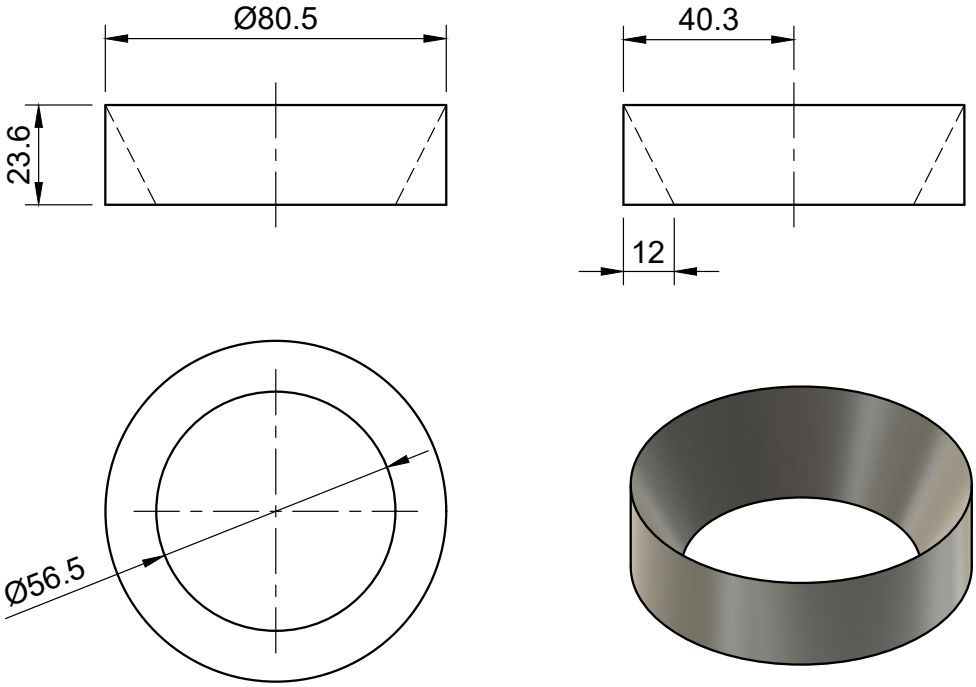
Created by	Emilia Szymańska	Project name	Ramp
Date	30.08.2021	Element name	Servo column
Material	PLA	Scale	1:1
Units	mm	Drawing No	16



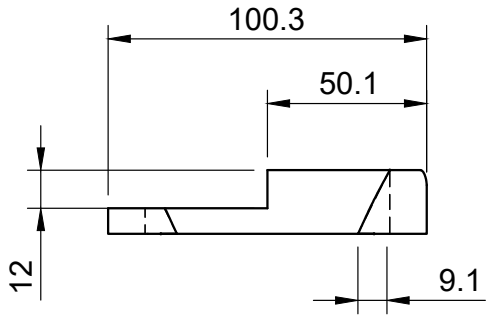


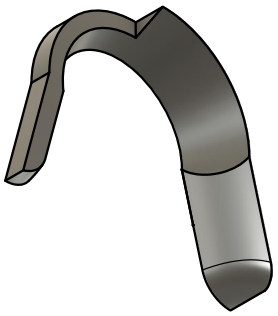


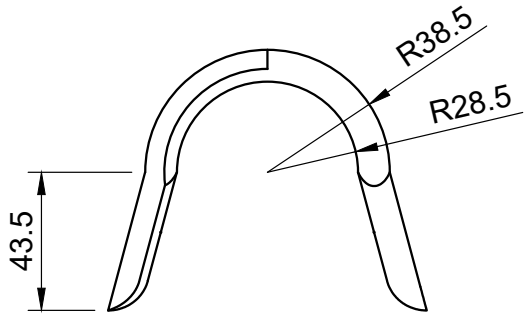
Created by	Emilia Szymańska	Project name	Cup rim
Date	30.08.2021	Element name	Cup rim
Material	PLA	Scale	1:2
Units	mm	Drawing No	17

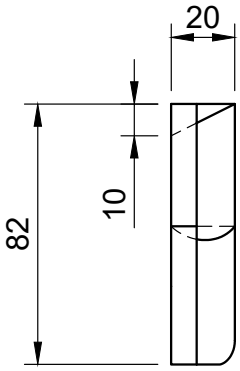


Created by	Emilia Szymańska	Project name	Positioning ring
Date	30.08.2021	Element name	Positioning ring
Material	PLA	Scale	1:1.5
Units	mm	Drawing No	18

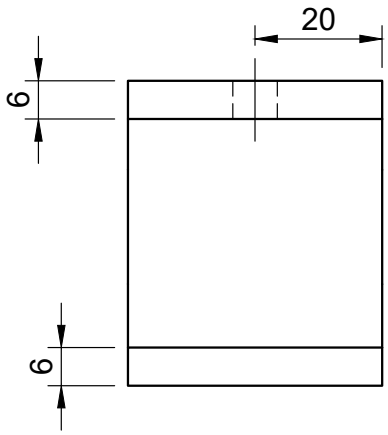


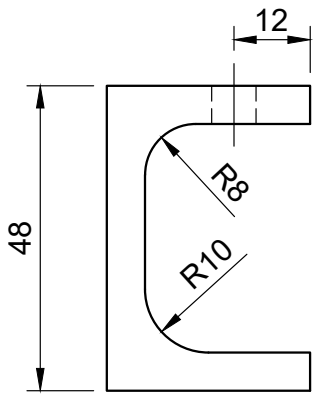


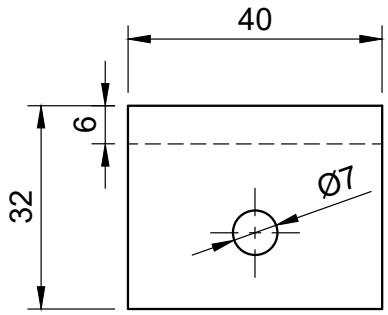


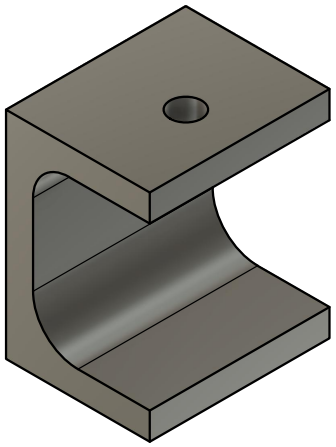


Created by	Emilia Szymańska	Project name	Cup positioner
Date	30.08.2021	Element name	Cup positioner
Material	PLA	Scale	1:2
Units	mm	Drawing No	19









Created by	Emilia Szymańska	Project name	Camera stand
Date	30.08.2021	Element name	Camera stand
Material	PLA	Scale	1:1
Units	mm	Drawing No	20