Preface

The De Vinci Innovation Center (DVIC) is a community of makers that develops technologies within philosophical and critical frameworks to shape our societies' futures. The objective is to implement real-world solutions as well as design projects to enhance public engagement, improve education, and overall provide scientific knowledge. Our researchers contribute actively to top-level international research in multiple fields, including artificial intelligence, human-computer interactions, education, and ecology. We believe that these objectives require a transdisciplinary approach, that bridges the gap between sciences, techniques, sociology, and philosophy. This is performed by collaborating with other scientists and industrial and startup sharing our values, to form strong research partnerships...

The Artificial Lives group, led by Dr. Clement Duhart, aims to develop the next generation of machines and Human-Machine Interfaces. The group members strongly believe that through the combination of Design and Engineering, human-centered technologies can blend into our environments to become invisible, vastly improving daily lives. To achieve this vision, the members contribute to human-computer interactions, cognitive enhancement through new forms of extended intelligence, learning platforms, and cobotic. Our bio-inspired, multidisciplinary approach couples AI and virtual reality with intelligent materials, robotics and the Internet of Things.

For the past two years, De Vinci Innovation Center (DVIC) students following the Creative Technologies curriculum had the opportunity to develop their vision on technology, innovation, and society. This proceeding is a composition of six master's theses, ranging from Machine Learning, Human-Computer-Interaction to Robotics. The authors strongly believe that developing alternative futures requires new types of engineering that take into consideration both the people's needs and the environment. These documents have been written to reflect this vision and refined over several months with an iterative reviewing supervised by the Principal Investigators.

The Authors, the Principal Investigators and the whole DVIC community is proud of releasing this first proceeding. We dedicate this first edition to Pascal Brouaye and Nelly Rouyres, without whom nothing would have been possible.

DE VINCI INNOVATION CENTER

Artificial Lives

DE VINCI HIGHER EDUCATION

List of Theses

Adrien LEFEVRE -

Adrien: Gamification of Interfaces for Learning and Performance Enhancement

Adrien LEFEVRE

Contents

Robotics middleware 1

1.1 Introduction

Robotic systems require complex infrastructure to be able to operate correctly. With numerous different parts working together that have to communicate and coordinate, these systems need a robust way to enable such communication. At the same time, translating user commands into instructions that can be understood by the inner electronics of the robot is a complex process that should only be exposed to qualified operators.

Robotics middleware is a type of software that provides utilities to software applications used in complex robot control systems. It is sometimes referred to as "software glue" because it links together different software components to create a complete system. Using a robotics middleware is a solution to the problems defined above: the middleware facilitates communication, provides a way to package software in a structured manner, and enables hardware abstractions to make development easier.

In this chapter, we introduce ALFRED as a robotics middleware. It follows the structure of UNIX-like operating systems, with separation of privileges between system programmers and normal users. It enables communication between software components with an external message passing system. It allows robotics application designers to create interactions with robotic arms without the need for specialized education or to learn the inner workings of a robotic arm. Finally, it provides access to software utilities and a hardware abstraction layer to simplify the surrounding parts of developing robotics applications.

1.2 Related work

ROS [**ros**] was explored to use as a robotics middleware. ROS is a set of software pieces to help the development of robotics applications. It provides hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. It enables two-way communication via topics. Components can subscribe to topics to receive messages, but also publish to topics to send messages. Software comes in the form of packages, which are units that can be run, used as dependencies or for configuration.

Another middleware is ISAAC, a solution by NVIDIA specialized for, but not limited to, high compute situations. ISAAC links together software in a graph structure, where nodes are the computational software pieces and edges are the exchange of data between the nodes. Custom applications can be created to implement functionality to robotics systems and be integrated into ISAAC as nodes. ISAAC also gives access to tools for simulation and training of deep learning models. With ISAAC GEMs, users can add AI-enabled data processing with pre-trained models.

1.3 Architecture overview

ALFRED is built around a publish/subscribe model database that enables inter-component communication. The middleware follows the structure of UNIX-like operating systems, with a Kernel and a Userspace. Robotics applications developed by the users run in the Userspace, while critical components run in the Kernel. The Kernel and Userspace are separated by the Pub/Sub messaging system, which creates a clear border between the two layers of the middleware. We use this clear border to manage the software's permissions over the hardware: we disable hardware access to the Userspace and only make the hardware accessible by the Kernel. To access data from the hardware, the Userspace has to

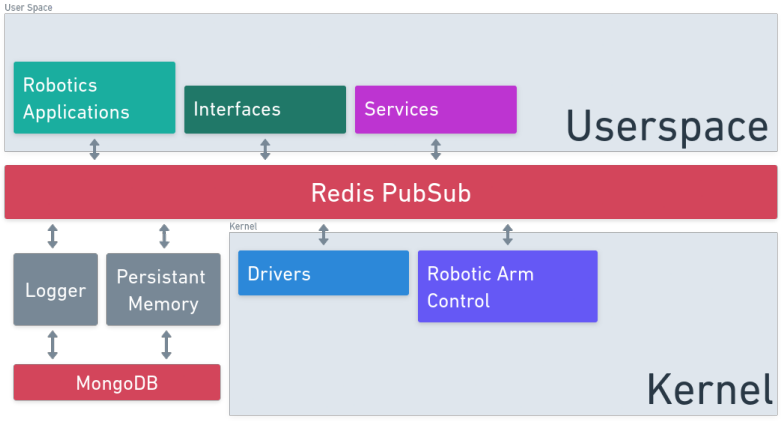

Figure 1.1: ALFRED's architecture

go through the Kernel first, by using the Pub/Sub. Our middleware's architecture is demonstrated in figure 1.1.

The middleware runs in Docker [**docker**] and is deployed with Docker Compose [**docker-compose**]. Each component is deployed as a separate container. Using containers to deploy our software makes it easier to separate the different parts of the system but also handle dependencies. Networks[1] are used to enable communication between components selectively. We use environment variables to configure global behavior at runtime, e.g. if the real arm should move or not.

1: Networks are part of Docker and enable containers to communicate with each other whereas they would normally be isolated.

Kernel The Kernel is the core of the middleware. It has complete control over the robot arm. It contains a Hardware Abstraction Layer (which we call the drivers) and the robotic arm control software. Drivers are the interface between external devices, such as cameras, and the Kernel. They can get data from and send commands to the devices. On the other hand, the Robotic arm control component contains the Application Programming Interface (API) for moving the robotic arm, as well as utilities to manage its state (starting, stopping, handling disconnections. . .).

Userspace Userspace contains user-created components. To house user-created software, we developed the Robotics Applications Component (RAC), the interfaces

and the services. The RAC contains applications, which are pieces of software that interact with the arm and implement the functionalities. Applications are modular: they can be developed outside of our middleware and integrated later with minimal changes thanks to a plug-and-play structure. Interfaces are ways to visualize what is happening in the system and interact with it in a user-friendly way. Services are components that are used to compute data but do not interact with the arm, like deep learning algorithms. They can also be used by multiple applications.

Inter-component communication Components communicate with each other using a Pub/Sub messaging system. This component sits between Kernel and Userspace and prevents them from interacting directly with each other. Instead, all communication has to go through the Pub/Sub. It provides a clear separation between Kernel and Userspace, and is our implementation of protection rings, or hardware access restriction. The Kernel is at ring 0 (no restrictions, most privileged) and Userspace at ring 1 (more restrictions, less privileged).

1.4 Kernel

1.4.1 Related work

Robotic arm control

Most robotic arms have an API exposing functions that allow users to control the arm more easily. Instead of controlling each of the arm's motors individually, the API exposes functions to execute complex movements, such as moving the arm to a certain position in space in a straight line, or drawing a curve from one point to the other with a certain radius.

Internally, the robotic arm's controller still translates user commands into individual motor movements. Most robotic arms provide a way for experienced developers

to send these individual motor movements directly to the arm, thus reducing the load on the onboard controller. These movements can be obtained manually via mathematics, but software like PyBullet[**pybullet**] or MATLAB[**MATLAB**] makes it easier. They provide utilities to specify robot-specific parameters, like the number of degrees of freedom and the length of the links between the motors, from existing data. Without these tools, the programmer would have to compute these parameters themselves. They can also provide access to simulation, creating a digital copy of the robotic arm to predict behavior ahead of time.

Drivers

In the Linux kernel, drivers are implemented as modules which can be added or removed. They are written in C or Assembly for low level control over the hardware, and they expose a file which presents the data from the hardware once it has been acquired and treated. Each driver exposes a set of functions which can be called to generate data or interact with the device.

In ROS, drivers are implemented as packages, like any other software piece, that produce data to a topic. These drivers are one or more layers above the raw hardware drivers of the Linux kernel. They often use libraries which read from the device files and expose this data in a more readable format. ROS drivers are flexible in that they don't need to stream all the data all the time; instead, they can be made to output data only when something significant happens, like when a spike of current is detected from a sensor, so they can arbitrarily abstract the data for the target software.

1.4.2 Robotic arm control

The Robotic arm control component is implemented in Python 3.9 and is deployed as a container. The component is tasked with receiving commands and data from the whole system and moving the robotic arm. Its

algorithm is described in 1.

Algorithm 1: Robotic arm control

```
1 while true do
2     message ← message from pub/sub;
3     if message is not empty then
4         parse message;
5         if message is a command then
6             execute message on robot arm;
7         else if message is data then
8             save and use message;
```

1.4.3 Drivers

Drivers are implemented in Python 3.9 and are deployed as a container. The container has full access to the host machine's hardware to be able to read and manipulate external devices. In the container, each driver is started as a background process, allowing it to run concurrently with the other drivers. When every driver is started, the parent process runs an infinite loop to keep the drivers running. Drivers have two main components: the `DataProducer` and the `CommandGetter`, which are started as threads and run indefinitely. They communicate with the rest of the system via the Pub/Sub. Topics used by the drivers have a special nomenclature, which is `device-data-<device name>` for the `DataProducer` and `device-command-<device name>` for the `CommandGetter`. Here are the algorithms for the `CommandGetter` 2 and the `DataProducer` 3:

Algorithm 2: CommandGetter

Data: *topic* ← "device-command-<device name>"

```
1 while true do
2     command ← command from pubsub;
3     if command is not empty then
4         parse command;
5         send command to device;
```

Currently, drivers are implemented for these devices:

Algorithm 3: DataProducer

```
topic ← "device-data-<device name>";
while true do
    data ← data from device;
    if data is not empty then
        publish data to topic;
```

- Realsense D435 cameras
- BLTouch sensors with a microcontroller as an intermediary data via Serial communication
- Microphones

The behavior of the drivers can be customized by environment variables set in the Docker Compose file.

1.5 Userspace

1.5.1 Related work

Robotics applications

In ROS, applications can be launched via the command line, but also programmatically using the roslaunch API. This requires to create a launch file specifying parameters for the application.

Outside of ROS, programs often expose APIs to make interaction possible. These API use protocols to structure communication, mainly REST (Representational State Transfer) and RPC (Remote Procedure Call). RPC is suited to executing applications from outside but has limited compatibility with the web. Thus, we considered three REST API frameworks from Python:

- Flask
- Django
- FastAPI

Flask is a minimal web framework which contains basic features such as routing, debug and static files.

Django is a more heavyweight framework for creating

complex websites with an SQL database as its centerpoint. It follows the Model-View-Controller pattern and has many features including an auto-generated admin panel.

FastAPI is a web framework built on top of Starlette, an ASGI (Asynchronous Server Gateway Interface) framework/toolkit. FastAPI has all the features of Flask with added functionality, such as OpenAPI docs generation and async support.

1.5.2 Robotics applications

The robotics applications component (RAC) is implemented in Python 3.9 and is deployed as a container. The RAC gives access to the robotic arm control component to the user. The RAC and the robotic arm control component communicate with each other via the Pub/Sub. The API uses FastAPI [**fastapi**] as its main technology. The RAC contains the applications which implement functionality on the robot arm. Applications can be called or interacted with via HTTP requests. To manage applications within the software, we created a class called `Apps` class, which is a subclass of Python's `multiprocessing.Process`. The `Apps` class provides utilities to start applications, check if they are running, handle exceptions and provide two way communication between the application and the RAC.

Applications are managed by the RAC's `ContextManager`, which is tasked with starting and stopping apps. The `ContextManager` only allows for one application to run at a time, to avoid conflicts if two applications wanted to move the robotic arm at the same time.

Applications are separate processes with their own separate memory, instead of threads that share their memory with their parent. This helps avoid errors with memory management, and bypasses the restrictions of some libraries used, e.g. OpenCV not being able to show windows in anything other than the main thread.

Applications can be developed outside of our middle-

ware and later integrated thanks to the structure of the RAC 1.2: applications are kept separate from the rest of the API, and live side by side. To expose an application, a route[2] needs to be created. The route is then added to the applications router[3] which is itself added to the main router.

2: HTTP method (`GET`, `POST`, ...), path (ex: `/applications/grasping`), and handler combination.

3: Part of the software that determines which route receives a specific request.

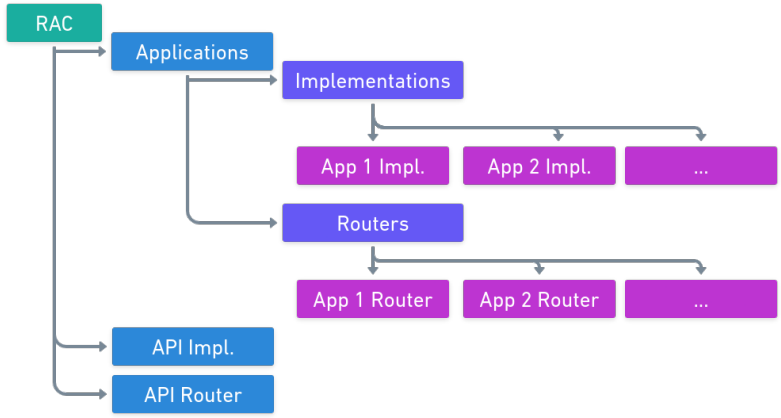

Figure 1.2: File structure of the RAC (simplified)

Evaluation

Students were tasked with developing applications using ALFRED. One of them created a driver for a BLTouch bed leveling device interfaced with a micro-controller, and an application using this device to create a 3D map of any mostly flat surface. Another created an interface plugging into the middleware to act as a dashboard, showing system information and executing applications. Here were their remarks about the system:

- The application was able to be developed outside of the middleware and then integrated with minimal changes, in less than a day.
- Creating the application didn't require extensive knowledge about robotics or the robot arm.
- The driver was conceived with few (about 30 added) lines of code from a basic template.
- The data exposed by the RAC allowed the dashboard to show meaningful system data such as the current position of the robot.
- The dashboard was able to be developed outside of the system and then integrated only by setting

up the appropriate Docker environment.

1.5.3 Services

Services are implemented as Docker containers. Since they are a bit more generic, the user is responsible for specifying the dependencies for their services in the component's `Dockerfile`. Examples for services are a Natural Language Processing server or an API for external programs such as Unity. We separated services from robotics applications because some services could require different versions of Python, or even a completely different runtime environment. At the moment, services have to be added manually into the whole middleware's Docker Compose file.

1.5.4 Human-Robot Interfaces

Interfaces are a type of service meant to handle input and output from the user. An example of interface is a Web dashboard for viewing the status of the system and managing applications.

1.6 Inter-component communication

1.6.1 Related work

Message handling can be done with ROS via its topics. Messages can be sent by pieces of software to a specific topic, and other pieces of software can subscribe to topics to get the messages. Messages are defined by a `type` which defines the structure of the message and allows ROS to automatically parse messages.

Redis is another solution for realtime communication between processes, specifically its Pub/Sub module. Redis Pub/Sub uses channels where publishers can push data and subscriber can subscribe to get data. Messages don't have types which means the publisher is responsible for encoding the data and the subscriber is

responsible for decoding it but the communication was found to be faster than ROS [**redis_benchmark**].

Other solutions for inter-process communication include D-Bus. D-Bus is based around a message bus daemon which applications connect to. Applications use a library, libdbus, to interact with the message bus. The daemon acts as a router, sending messages from one application to another. Communication is one-to-one only.

1.6.2 Communication bus

Inter-component communication is done with Redis [**redis**], specifically Redis' Pub/Sub implementation. It runs in a Docker container from the Bitnami image, `bitnami/redis` version 7.0 with default settings. The Pub/Sub sits in-between the Kernel and Userspace and allows components to send each other data.

1.6.3 Persistent storage

The Redis Pub/Sub doesn't persist the data it receives to disk, so MongoDB is used for long-term storage. Data stored includes logs and component-specific data, but it can also serve as a way to save messages sent via the Pub/Sub. The Kernel possesses its own database for storing system-critical data, while the data from the Userspace database can be accessed by both Userspace and Kernel.

1.7 Conclusion

1.7.1 Applications

In this chapter, we described an infrastructure for developing applications on robotic arms. It follows the architecture of UNIX-like operating systems with its separation into Kernel and Userspace and the associated protection rings to define privileges.

This middleware makes developing robotics applications accessible to users with limited knowledge about robotics and robotic arms. It means that developers need less training and that they can have more confidence that their software won't cause damage on the system.

Thanks to its modular architecture, software can be created outside of the system and integrated later with minimal modifications. This is useful since debugging becomes harder the more complex a system is. With isolated software, developers can find and fix issues more rapidly.

With the use of a central Pub/Sub messaging system, components don't need to know who they are sending their data to, but only need to determine the name of a channel that the relevant software can listen on. With this, we can reduce how strongly components depend on each other, which is called coupling. Reduced coupling makes breaking changes less frequent when modifying related code since the components communicating are not directly linked to each other.

1.7.2 Limitations

Running every component as Docker containers adds a complexity that would not exist with native execution. The robotics applications component still has to be run in `privileged` mode[4] , because some applications require access to the screen to show information. Running in `privileged` mode diminishes the usefulness of having a separation between Kernel and Userspace, since in the end Userspace can still access devices directly.

4: Privileged: has access to every device. Unprivileged: cannot use the host computer's devices.

Running in Docker also means components are difficult to debug once they have been integrated into the system. This is because every time a modification has to be done, the system needs to be stopped, built, and then started again for the changes to be taken into account. Approximate times are shown in table 1.1. Build times can be especially long when dependencies are changed and the whole image has to be rebuilt. Setup times

can also be long depending on the applications, e.g. when loading a deep learning model. Such long restart times decrease the efficiency at which the project can be iterated on.

	Time (best, seconds)	Time (worst, seconds)
Stop	20	30
Build	20	600+
Start	20	30
Setup	10	300+
Total	70	960+

Table 1.1: System restart times.

As it is, the controller is using ALFRED's specific robot arm's API to move the arm. This means that applications developed for this specific robotic arm will not be compatible on another arm. It also means that the control of the finite movements of the robotic arm comes from an external program, and so the middleware's controller component can't directly influence said control.

1.7.3 Future works

Issues with the API running in `privileged` mode can be solved by delegating the responsibility of showing information to the Kernel. An interface could be implemented to get image buffers from applications in Userspace and then show these image buffers on screen. Options for configuration would need to be added to respond to the needs of applications in terms of interface. Also, since window controls would be under ownership of the Kernel, applications wouldn't be able to capture user input by themselves, so the Kernel would need to pass these inputs through to the applications.

Some methods were used during development to reduce restart times, but they are not properly integrated and impractical to use by the general public. We plan on creating a fully integrated `dev` mode for the system. `Dev` mode would enable programmers to make changes without having to rebuild the whole system. This mode would add a feature to automatically reload the Docker container's program(s) when a file is changed or upon

user request. Files in the host computer would need to be synchronized with the files inside of the containers, which can be done with Docker `volumes`.

As they are implemented currently, services and interfaces are not very well integrated into the system. Indeed, the user has to create their Docker image on their own, add it to the main Compose file, and configure it properly. We want to make services and interfaces easier to add to the system by automating the discovery and integration of services into ALFRED. This piece of software would expose utilities to add new entries in the main Compose file securely with a template. It would allow experienced users to provide their own Docker images, program files, and configuration manually, and less experienced users to use common configurations to quickly and easily deploy new services.

We plan on generalizing the middleware for every robotic arm. To do so, we could first create a mapping between our generalized API for robotic movement and the associated function for every robotic arm. This would change the function that is called within robotic arm control depending on the model of the robotic arm used, but wouldn't change the function called in Userspace. Later, we would step away from using the robotic arms' APIs, and compute the robot's movements ourselves using Inverse Kinematics, only sending motor movements to the robotic arms.

Human-robot interaction 2

2.1 Introduction

As robotic arms evolved from specialized operators capable of only executing a single task to complex structures with high computing power, humans became able to interact more closely with the machines.

Human-robot interaction (HRI) studies the means of communication between humans and robots by designing, understanding, and evaluating robotic systems that coexist with humans. It is a multidisciplinary field with contributions from human-computer interaction, artificial intelligence, robotics, natural-language understanding, design, and psychology.

Interaction can be separated into two general categories: remote and proximate. In remote interaction, the human and the robot are separated spatially, or even possibly temporally. Remote interaction can be `teleoperation` if the robot is controlled by its interaction with the human as a general supervisor, or `telemanipulation` if the human controls the robot completely via a physical manipulator.

In proximate interaction, communication is often more social, focusing on more abstract concepts such as emotions. The robot can perceive the human with sensibility to touch, for physical interaction; sight, for analyzing posture and facial signals; or hearing and speaking to communicate verbally like a real human.

We designed two interfaces for remote and proximate interaction. First, hand movement and gesture control, which allows the human to manipulate the robotic arm remotely by using their hand as a guide for the robot and making hand gestures to trigger actions. And second, voice control, a proximate interaction using natural language understanding to enable the robot to accept

commands formulated in a natural, "human-like" manner.

2.2 Related work

Many technologies can be used to develop interactions due to the flexible nature of robotic platforms. From computer vision, to wearable electronics, to virtual reality, many things can be adapted to enable control of a robotic arm.

Using a Virtual Reality headset, Lipton et al. [**robot_control_vr**] were able to create a platform for teleoperation, implementing tasks such as pick and place, assembly, and manufacturing. It takes inspiration from Penfield's Homunculus model [**penfield1950cerebral**], creating a mapping from human to robot enhancing sight and hand feeling.

Complete structures can also be constructed to place the user in a sort of "command chair", where they can precisely move the robot [**centauro**] 2.1.

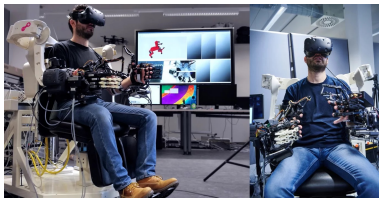

Figure 2.1: Centauro project control chair

To create a direct link between the robot brain and the human brain, Salazar-Gomez et al. [**robot_control_eeg**] use an EEG (Electroencephalogram) headset to detect brain signals related to unexpected error making (ErrPs) to correct mistakes in classification tasks. It can change the behavior of a robotic arm when it is making a mistake during a task consisting of sorting object into different labelled bins.

The body itself can act as a control interface using wearable devices [**robot_gesture_control_wearable**]. By measuring biceps, triceps and forearm activity, but also arm motion, DelPreto et al. were able to control a drone through obstacles using only their arm.

Interfaces can be used to explore the relationships between human and robot. By studying the emotional response of humans giving an object to a robot, and the robot taking the object in different manners, we can, for example, find the characteristics of robotic move-

ments that make them more trustworthy in the eyes of humans, to make them more socially apt, and to mimic human-human interactions [**robot_handoff**].

There are also ways of controlling a robot that don't involve physical contact or devices. Matarneh et al. [**matarneh_maksymova_deineko_lyashenko_1970**] used a deep learning algorithm, specifically a multi-layer perceptron, to interpret voice into commands. With these commands, researchers were able to control the movements of the robot directly from voice data, while other solutions involve translating voice into text first, and then text into commands [**Janicek_2021**].

2.3 Hand movement and gesture control

2.3.1 Introduction

A robotic arm closely mimics the workings of the human arm, with joints and muscles in the form of motors. It is not surprising, then, that we would want to create a link between the human arm and the robot arm. On another level, grasping tasks are some of the most common tasks of a robotic arm, just like the hand of the human.

These are the reasons why we created hand movement and gesture control for ALFRED. By mirroring the robot and human arms, we can create a strong connection between machine and operator. We can take advantage of the dexterity of the hand and translate this dexterity into the robotic arm by interpreting complex hand gestures into commands for the robot. Hand movement and gesture control can be used as a remote interface, making the robotic arm act as the user's arm in another place, or as a proximate interface, acting like a mirror showing a robotic image of the user to aid them in their current task.

2.3.2 Contribution

Concept

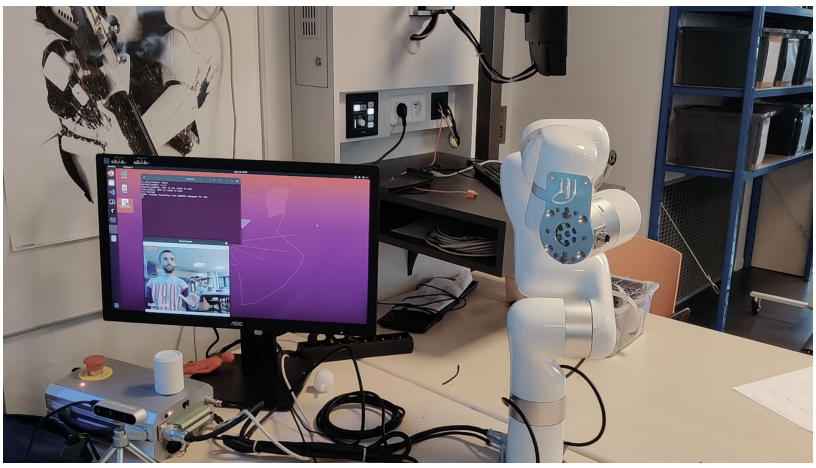

Figure 2.2: Operating the arm with hand control. On the left, the computer running ALFRED with a camera pointed at the operator, showing the annotated camera feed. On the left, the robotic arm being controlled.

Hand control allows an operator to control the robotic arm with their hands 2.2. It uses a camera pointed towards the operator and a hand recognition algorithm to detect the position of the hand. It then interprets the data to control the robotic arm. The hand control interface moves the robotic arm by comparing the position of the center of the hand[1] to the center of the video frame. If the hand is on the left of the center of the frame, the robot will move left (as seen from the front). The further away from the center the hand is, the faster the robot moves.

1: Taken as the middle of the segment between the wrist and the index finger metacarpal

We trained machine and deep learning models to detect gestures which allow for further interaction, such as opening and closing a gripper, or starting and stopping other applications.

Implementation

Hand control has been implemented following the principle of dependency injection. The software contains only video processing, while video acquisition and moving the robotic arm itself are left to system developers. This allows modularity and adaptability for many, if not all, systems. In ALFRED, video acquisition is done through the drivers with a stream from the Pub/Sub abstracted by a class. Moving the robotic arm is done

through a function injected into the Hand control software. The function takes in landmarks and gesture data from the model and translated them to robot commands specific to the robotic arm.

For the processing part itself, Hand control uses video frame data, hand detection, and post-processing to transform hand position into robotic arm commands. From frame data, hand detection is run using Mediapipe Hands[**mediapipe_hands**].

We extract 3D landmarks which represent the various points of interest on the hand. These landmarks represent the wrist and each finger joint, as well as each finger tip, for a total of 21 landmarks.

Two models were built to detect gestures from the landmarks. We built a dataset of hand gestures and their corresponding landmarks. The data was collected by taking a video of a subject doing the same hand gesture for a couple of seconds, moving and rotating their hand constantly and randomly. We only recorded the landmarks instead of the whole frames, so we only had 42 (21 landmarks, and x and y values for each landmark) values to record instead of the 640x480x3 values of the RGB frame. The data was augmented with small random rotations, as well as mirroring to enable generalization for both hands. We then trained a Random Forest using scikit-learn[**scikit-learn**], and a Multilayer Perceptron (MLP) implemented in PyTorch, with a few layers activated by the ReLU function.

From the hand landmarks, we create robot commands. Our initial command method was to create a linear mapping between the position of the hand and that of the robot's end effector. User feedback allowed us to finetune the method to be more intuitive and capable. This second method mimics the working of game controller joysticks. We transform the camera fram into the joystick: when the user moves their hand to the left of the frame, the robot moves continuously to the left. The speed of the movement is dependent on the distance between the center of the hand and the center of the frame.

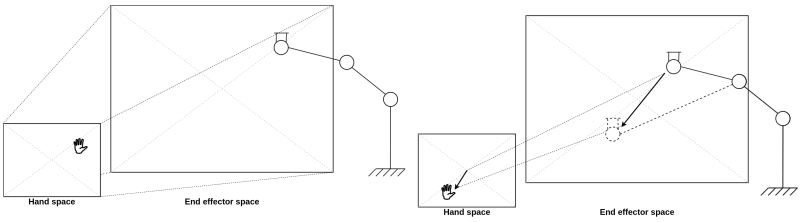

Figure 2.3: First (left) and second (right) methods for translating hand position to robot command.

The center of the frame contains a dead zone a couple of pixels in radius. When the user places their hand in this zone, no movement command will be issued. Gesture commands can still be recognized and issued. This was found to be necessary because the arm would make small movements when the user didn't want it to move. It would also add latency to the next movements: robot commands would be queued continuously and clog up the command queue.

To implement this control method, we compare the center of the hand to the center of the frame, and multiply the difference by an offset, found empirically (in our case, it is `0.022`). The commands are then sent to move the robotic arm. The two methods are illustrated here 2.3.

Evaluation

The goal of this interaction was to create a way to control a robotic arm that would be more intuitive than using gauges and dials on a computer screen. We selected four criteria to measure its effectiveness:

- Intuitiveness: how natural it feels
- Usability: how good it feels to use
- Mobility: how it can move in its environment
- Precision: how difficult small movements are to achieve

We asked 8 participants to test the interaction. Of the 8, 5 had prior knowledge of the system and had at least a basic understanding of robotics, while the rest had no prior knowledge and no knowledge of robotics.

We used two tasks to evaluate our interaction:

- Free exploration (intuitiveness, usability, mobility)
- Moving a plastic can balanced on top of a piece of wood (precision) 2.4

Figure 2.4: Preparing to grab the piece of wood to move the plastic can

Free exploration was overall appreciated by the participants. It took them about five minutes to understand how the system works, which we found to be acceptable. Six out of the eight found the chosen method to be intuitive, and all the participants found it to be an effective method. There was one problem when starting the control session, specifically with the initial hand pose: the hand had to be perfectly centered or else the software would send the robot commands when it was still setting up, resulting in erratic movements that would sometimes break the system. Once aware of this problem, participants were able to avoid it reliably. The most limiting factor was range of motion, where the arm would make very big and fast movements when approaching the end of its operating zone. This would oftentimes break the software which would require a reboot. We can therefore validate intuitiveness and usability, but there needs to be further work on mobility by making further regions safer to operate in.

Our precision task used gestures, namely closing and opening the hand, to grab and release with the gripper robotic arm's gripper. The objective was to pick up and move a piece of wood with a 3D-printed soda can balanced on it in a circle, then put it back down.

Four participants, all with prior knowledge, managed to complete the task, while the other four failed. Of the four that failed, two did because the gripper released the piece of wood too early, causing it to drop to the table. From this experiment, we can conclude that precision tasks required more trained operators, and that we had to improve the reliability of the system, specifically with hand gestures.

2.3.3 Conclusion

Applications and use cases

Hand movement and gesture control can be applied to any robotic arm to allow a minimally-trained operator to control the arm.

With this more natural method of control, as opposed to moving a slider or clicking on arrows, operators are able to achieve higher precision and a stronger connection between them and the robot.

More precise control for minimally trained operators means Hand movement and gesture control can be used by factory workers to move heavy weights, for example in a construction of metal factory, with a bigger industrial arm. It can also be used by researchers in laboratories, for moving dangerous pieces of equipment or chemicals during an experiment where hand manipulation, even through a glove box, would be risky. Finally, given the appropriate equipment, we could imagine a robotic arm moving through rough and dangerous terrain in rescue or cleanup operations.

Limitations

The main limitation for Hand movement and gesture control currently is the dimensionality: the arm can only be made to move in a plane. It cannot move in three dimensions which severely reduces its capabilities. Even more importantly, there is no way to rotate the arm. It will always be facing in the same orientation.

Secondly, using such a simple interface as the hand means that more advanced controls need to be hidden behind certain commands, or in our case gestures. This increases the complexity for understanding the system.

Future works

In the future, we would like to improve the system on the problems found in the user study and alleviate one of the limitations.

Mobility would be improved by creating safeguards around the edged of the range of motion to create smoother movements around these edges. Range of motion would be reduced a bit but to the benefit of more system reliability and easier handling because the system would not break or create sudden unplanned movements.

Grabbing reliability, or more generally gesture handling reliability, also needs improvement. Systems can be put in place to avoid suddenly releasing a grabbed object because of an error in gesture detection. For example, with buffering and outlier removal.

Another main aspect that would be improved is the dimensionality. While the arm can currently only be made to move in two dimensions (in a plane) with Hand movement control, we would like to make it move in three dimensions. This could be done by using depth frame data to measure the proximity of the hand to the camera. With some calibration at the beginning of handling, we could detect wether the operator moves their hand backwards or forwards, creating the possibility for the arm to move in the three dimensions.

2.4 Vocal command control

2.4.1 Introduction

Recent years have seen the rise of home assistants such as Siri from Apple or Alexa from Amazon. These assistants use complex deep learning algorithms to interpret the sound of your voice into commands to execute. These devices allow their users to control their home and automate tasks without having to touch anything. They act like a butler, constantly listening for requests and executing them when received.

We wanted to reproduce the same functionality for ALFRED: to allow users to talk to the platform as a means of interaction. Like home assistants, the objective was to create a "butler" that listens, like a human, to commands expressed in natural language, instead of pre-defined sentences. It would thus be more integrated into its environment as something almost living and intelligent.

2.4.2 Contribution

Concept

Voice control allows an operator to control the robotic arm vocally. With data from the microphone driver, a Speech Recognition algorithm interprets the voice into text (Speech-To-Text, or STT). Text data is then interpreted by a Natural Language Processing (NLP) algorithm to extract intent and data from the command. From the intent, the system determines which API function to call, also sending the extracted data when necessary.

Implementation

We built a driver for microphones which provides raw audio data but is also capable of doing Speech-To-Text (STT). Using Azure Speech [**azure_speech**], we convert raw sound data (also from the driver) into text. We do

continuous speech recognition: the STT model recognizes each word as it is spoken, and combines words together when a pause indicating the end of the sentence is detected. At the moment, only complete phrases are sent by the driver.

With the sentences detected by the STT part of the microphone driver, we run an NLP algorithm. Voice control uses RASA[**rasa**], a machine learning framework for building automation around text conversations. It uses Natural Language Understanding (NLU) to process text into intents and variables, which we call command data.

Intents are the purpose behind the sentence that was treated. Currently implemented intents include:

- `call`: Activate listening. Unless the system is called, it will not respond to commands, to avoid doing everything it hears against the operator's consent. Listening stays activated for a short period of time or until it is manually deactivated. An example of a sentence for this intent is: "Hey Alfred".
- `uncall`: Deactivate listening. An example is: "Thank you Alfred".
- `grab_object`: Grab the specified object. An example is: "Can you grab the cup ?"
- `call+grab_object`: Associates the `call` and `grab_object` intents. An example is: "Hey Alfred, grab the bottle please".

Command data is a word or a set of words extracted from the command, that can change for the same intent. In the case of grabbing objects, data would be the object to grab. For example, in "Grab the pen please", RASA would extract the intent as "grab" and the data as "pen".

From the intent, the system determines which API function to call, also sending the extracted data when necessary.

2.4.3 Conclusion

Applications and use cases

Voice control is used within ALFRED to interact with the system vocally. It allows users to manipulate the system in a natural manner, like talking to a human assistant. Voice control can also be integrated in other systems to make them controllable by voice. API routes can be mapped to voice control intents to make the system more accessible to users, in line with the objective of seamless integration.

Its applications can be imagined in makerspaces, as an interface for an automated personal assistant. The user could be doing a manual task and direct the assistant with their voice, in a natural manner. The user wouldn't need to put down their equipment to manipulate the assistant, and instead could continue doing their task in a seamless manner.

It could assist people who can't move their arms because of a disability who would have trouble controlling the robotic arm with a mouse and keyboard, or a controller, or even Hand and gesture control.

Limitations

Voice control is held back by its training complexity. Due to the nature of the software used, voice control cannot generalize easily for new commands or conversation paths. Each new intent needs dozens of training samples which need to be carefully selected, and each time the model must be re-trained and re-deployed. This makes it difficult for developers to extend Voice control with their specific use cases.

Future works

More intents and stories need to be developed to enable the Voice control software to be an autonomous assistant. Currently, it can only call functions from the robotics platform, but the goal is to make it more of an interface

that can show the full capabilities of the platform, from listing all available features to describing the current state of the platform, or even to make tutorials and demonstrate how to do certain tasks with or without the robotic arm.

2.5 Conclusion

One of our objectives with ALFRED is to allow people to design interactions with a robotic arm and to provide the tools to facilitate doing so.

We explored the possibilities of ALFRED by creating two interactions of our own: hand movement and gesture control, and voice control. These interactions allowed us to finetune our system to make it more usable by future developers. When designing the interactions, we created services to enable the creation of the RASA server, a software unit that was not a driver, but also not a robotic application.

We discovered the idea of creating a connection between man and robot by mapping body parts to the robotic arm, and we searched for ways of translating body movements into robot commands that would feel natural to us humans.

Finally, deep learning was used to transform a simple robotic arm into a smart assistant and an integral part of the space it is located in.

Robotics applications 3

3.1 Introduction

The actions a robotic arm can take decide its usefulness. To assist a person in their everyday lives, the arm should be able to, among other things, manipulate objects or interact with machines. To assist in an industrial context, it must be able to use the tools necessary for the task and to be reliable.

What we call applications within ALFRED are the pieces of software that implement the possible actions of the robotic arm. As we presented in a prior chapter, they live in the Userspace where any user is free to implement their applications for their use cases.

We present two applications that were developed to add capabilities to the robotics platform: writing and drawing, and object grasping and manipulation.

3.2 Writing and drawing

3.2.1 Introduction

Writing is a complex motor task that children take years to do properly. Drawing is even more complicated, and becoming an artist capable of expressing their thoughts in paper or a digital format is a lifelong endeavor.

A robot can learn these skills by learning the movements once without needing time to practice. Writing and drawing can be useful in automation, where a factory would want to create custom handwritten notes, or sign papers with a pen automatically. It can also help people who can't write by themselves, and don't want to, or can't, transition completely to digital media.

3.2.2 Contribution

Concept

This application uses the robotic arm to write text autonomously. It transforms its input text into a sequence of movements of the robotic arm, thus enabling the robot to write on any flat surface, independent of orientation.

The original gripper could not grasp the pen efficiently enough to do continuous writing, so we developed a custom mount for the pen, taking advantage of the provided schematics for the arm's end effector.

The system was then adapted to use the robotic arm to draw line art of images 3.1.

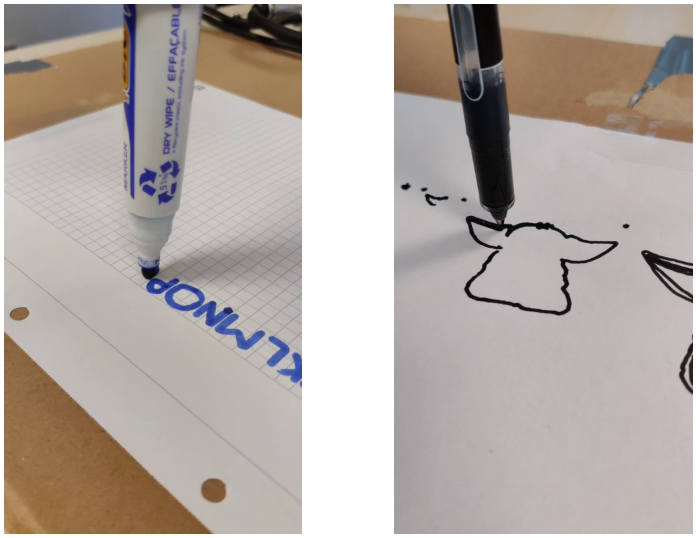

Figure 3.1: Writing and drawing

Implementation

Writing The API for the writing application takes in a string of the letters to write. It uses a mapping between letters and an array of robot instructions. These robot instructions are numbers and booleans, whose structure is adapted from the API of the robotic arm to be more comprehensible to the programmer. This structure is as follows:

- 6 numbers for x, y, z, roll, pitch, yaw. These values are not interpreted as absolute movements relative

to a fixed point in space but as relative to the actual position of the arm.
- 1 number for the radius of the movement. If the radius is 0, the movement will be a straight line. If it is bigger than 0, the movement will be a curve with the specified radius.
- 1 boolean to specify if the radius is in degrees or radians.

Here is an example for the letter D:

```
mapping = {
# ...
    "D": [
            # Starting pen up, go to the top left
    corner
            [10, 0, 0, 0, 0, 0, 0, False],
            # Lower the pen to the paper
            [0, 0, 5, 0, 0, 0, 0, False],
            # Draw the left bar of the D
            [-10, 0, 0, 0, 0, 0, 0, False],
            # Raise the pen
            [0, 0, -5, 0, 0, 0, 0, False],
            # Go back to the top left corner
            [10, 0, 0, 0, 0, 0, 0, False],
            # Lower the pen again
            [0, 0, 5, 0, 0, 0, 0, False],
            # Draw the semi-circle of the D
            [0, 7, 0, 0, 0, 0, 7, False],
            [-10, 0, 0, 0, 0, 0, 7, False],
            [0, -7, 0, 0, 0, 0, 0, False],
            # Raise the pen and prepare for the
    next letter
            [0, 10, -5, 0, 0, 0, 0, False],
        ],
# ...
}
```

Listing 3.1: Example for the letter D

The instructions in the mapping use a relative notation, but the performance of moving with only relative motions was not sufficient for the needs of the application. It added delays from calculating the absolute position internally and the movement was choppy. To correct this issue, we store the arm's position at the beginning of

each sentence and calculate the absolute position from the relative movements for each robot instruction.

Drawing Using the same system as writing as a base, we added the possibility for the arm to draw line art of images. This feature uses images in the Scalable Vector Graphics (SVG) format to extract and draw line art. Images in the SVG format are not described as a matrix of pixels, but rather a series of lines described by mathematical equations. This normally allows the images to be scaled up or down infinitely, but in our case, we use these equations as guides for the trajectory the robot should use to draw the picture.

We support a subset of the SVG format. The supported path instructions are:

- MoveTo, or M: move the cursor to the specified position without writing
- LineTo, or L: move in a straight line
- CurveTo, or C: move in an arc as described by a Bezier curve
- ClosePath, or Z: go back to the place the cursor started writing

SVG files are pre-processed by hand to remove all unnecessary information. Only the path attributes are kept, and they are formatted to have one command per line. In code, the pre-processed file is parsed into an array containing each command in order. The commands are then translated into the same type of robot instructions as writing. For circles and lines, we first translate the command into a mathematical equation which we then sample according to a manually defined, arbitrary precision level.

This translation essentially creates the same value as the alphabet mapping, where here the key is the id of the drawing and the value is a much more complex array of robot instructions.

3.2.3 Conclusion

Applications and use cases

Enabling a robotic arm to write allows anyone with such equipment to write or draw autonomously. This feature allows users to write text on paper without their hands, which can be useful for impaired users or in multitasking situations. This application can also be used to automate writing in a factory or small business. With drawing, company logos or signatures can also be automated.

Limitations

One of the drawbacks of our approach is speed. Due to the precise nature of writing, movements need to be slow or the letters produced by the arm are not well drawn, curved or slanted.

Another limitation is the complexity of the setup required. This is caused by the absence of any feedback mechanism on the pen and pen mount. The drawing surface needs to be on a perfectly flat and level surface because the pen cannot adapt itself to the surface. As an extension of this problem, careful calibration is required to make writing work. The pen needs to be placed precisely on the axis perpendicular to the drawing surface (Z-axis calibration). Too high and it will not make contact when writing, too low and it will be forced into the writing support and break. The plane of the robot end effector needs to be perfectly parallel to the plane of the writing surface, or it could progressively go away or towards the surface when writing (XY-axis calibration). This problem can be mitigated by using a large pen like a whiteboard marker and a soft writing surface, so that the pen can dig into the surface a little bit and still write, but it is not an elegant nor viable solution.

Future works

One of the possible improvements that can be done to this application is the automation of drawing. Currently, only SVG images are supported, and they have to be pre-processed manually. With edge finding and line fitting algorithms, we could transform any image into lines and then into mathematical equations that could be sampled to give robot instructions.

Another improvement would be the integration of hand writing. By mixing together drawing and writing, we could be able to interpret a user's hand writing as a drawing and use it in our alphabet mapping to replace the original simple letters by custom hand writing. This would improve the system for writing automation since the writing would be much more visually appealing. The system could then be used to automate the writing of custom sentences e.g. for notes.

Finally, to correct the limitation of the writing surface, we could integrate a bed levelling application (which was already developed on ALFRED) to find the location and orientation of the writing surface, removing the need for Z- and XY-axis calibration.

3.3 Object grasping and manipulation

3.3.1 Introduction

Robotic arms traditionally use hardware attached to their end effectors to interact with their environments. When a robotic arm is used to automate welding, for example, it will be equipped with a tool specifically made for this application.

However, in flexible environments such as homes or research laboratories, tasks are more varied than in assembly chains. Using the same approach as in factories would require tools to be changed too often, or specific tools to be made for every variation of a task or environ-

ment. Instead, by giving the robotic arm the capability to grasp and manipulate objects, we can make it adapt to many more environments and tasks.

3.3.2 Contribution

Concept

This application consists of looking for a specific object in a radius around the robotic arm and grasping it once it is found. The operator can then manipulate the object with the robotic arm through voice commands.

First, an object type has to be registered. This can be done at the application's start or during its execution. The robotic arm will then enter `scanning mode`: in a loop, it will look at its surroundings section be section to look for the object.

When the object is detected, the application enters `homing mode`. In homing mode, the arm centers the camera on the detected object. It starts off with big movements to get close as fast as possible, then uses smaller and smaller movements the closer it gets.

Once homed on the object, the robotic arm begins its approach for grasping. Using camera color frames, it checks that it is still centered on the object. Using camera depth frames, it detects how close it is to the object. Once it is close enough, it closes the gripper. Grasping force is estimated from the depth frames by finding the approximate size of the object. To verify the object has been grasped correctly, we look at the value of the current going to the gripper: a higher value than normal means the gripper is applying force to the object, and therefore that is has grasped it.

If the robotic arm grasped the object successfully, it can then await further commands via the voice interface. Possible commands are to move the object in a certain direction, to hand it to the operator, or to place it back on the experiment space 3.2.

If at any point in time the system loses the object, the arm

executes its `return sequence`, getting back into position for scanning.

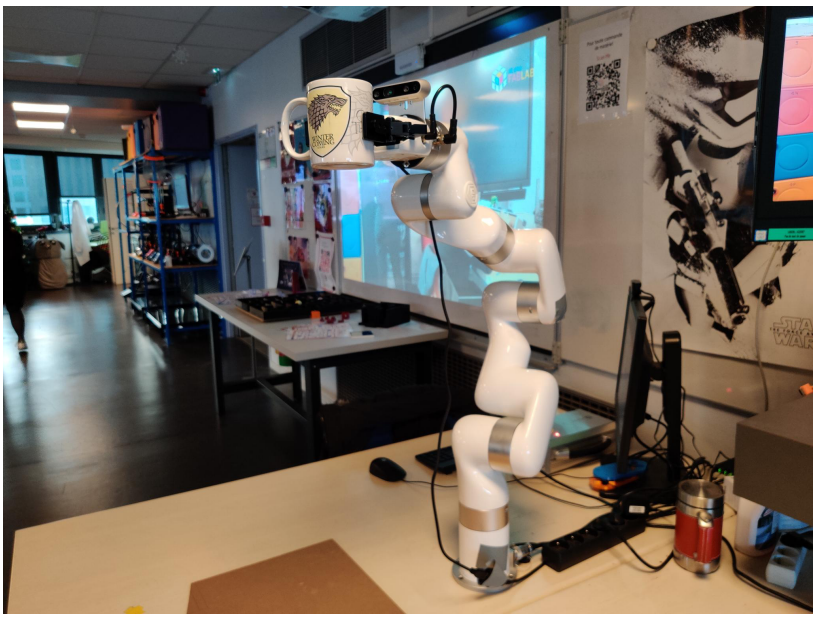

Figure 3.2: Robot arm grasping a cup, ready to accept commands for manipulation

Implementation

Starting and interacting with the application At its base level, the application is called via an API route within ALFRED. The object to grasp can be given when starting the application or left blank, in which case the arm will not scan for any object. The target object can be changed during execution with another API route. Within ALFRED, applications are run as processes and therefore cannot normally be interacted with[1] . We thus implemented two-way communication within the application using Python's multiprocessing Pipes. This allows runtime modification of grasping targets.

1: Contrary to threads, processes' memories are kept separate which renders memory sharing impossible and thus makes communication harder.

Image detection We use a deep learning image detection algorithm to detect the objects on the experiment surface. We chose to use YOLOv5[**yolov5**] because of its performance and the fact that only boxing was necessary for this application. YOLOv5 also enables the applications to run on hardware without GPU acceleration with smaller models that can run on the CPU, which increases compatibility and convenience for development. Color frames from the camera are processed by

the YOLOv5 model in a separate thread and are used in the decision loop of the application to search for and grasp objects.

Scanning mode Modes are determined by flags which are set or unset depending on the current mode, and change the behavior of the application in the decision loop. When the application is started, it automatically enters scanning mode. In scanning mode, the arm circles through a set of positions to search the experiment space for the target object. These positions are determined by linearly sampling values between two points. These points are the extremes of where the arm can move without colliding with its environment, specifically for its first joint. The result is that the arm moves in a semi-circle within the experiment space, and stops at each position for one second to scan for an object, in an infinite loop.

Homing mode When the target object has been found, homing mode is activated. Because the object can be anywhere within the camera frame when it is detected, it is necessary to place it in a known position within the frame to continue with the grasping operation. We call this operation homing. In homing mode, the system will try to center the target object within the camera frame. To do so, we calculate the difference in positions between the target object and the center of the frame, expressed with an `x` value for height and `y` value for width. We then calculate the necessary movement of the arm with a linear scaling. The movement of the arm is not always perfect, so we continue to home into the object until it is centered.

Grasping Currently, grasping mode and the behavior of the robot after grasping are not implemented.

Return sequence If the object is lost during homing or grasping mode, meaning the object moved, or there

was a detection error after the object was first detected, the robot needs to reset its position to begin scanning again. To do so, we execute a return sequence which places the robot back into its scanning position, and we also set the current mode back to scanning. The current rotation along the points defined for scanning mode is kept the same, so the robot can continue where it left off in its scanning after being interrupted by the detection of the target object.

3.3.3 Conclusion

Applications and use cases

A robot with grasping capabilities can be used at home, at work in research laboratories and FabLabs. With its ability to manipulate objects and tools, the robotic arm could help with tasks such as soldering, or screwing. It could automate repetitive manual tasks, such as product assembly, and assist in cleaning or sorting. It can either be controlled directly or be made to memorize the movements for a task after being directed once.

Limitations

Currently, object grasping is not yet implemented in the application. Work has been done to prepare this implementation however, and we found that grasping methods often require complex deep learning models to be reliable. We humans learn the mechanics of grasping by experience, but robots need to learn representations of objects to find where to grab. To obtain these representations, we need a large amount of data and powerful computers for training. The training and deployment complexity make it impractical to adapt to new use cases and situations.

The equipment can also limit the efficiency of the application: the gripper being a classic "pincer" gripper reduces the stability when grasping certain items in round shapes for example.

Future works

For the future, we plan to fully implement grasping and after-grasping behavior. This involves creating a basic algorithm for grasping with limited capabilities at first, and then using deep learning to teach the robotic arm to grasp more complex items. Once grasping is implemented, we can create a recording mechanism to memorize tasks to automate.

To solve the problem of the gripper, we could replace our pincer gripper with a soft robotics gripper using pneumatics. This type of gripper mimics the human hand more closely and is more adapted to more organic shapes.

3.4 Conclusion

ALFRED uses applications to implement functionality for the robotic arm. We developed two applications to demonstrate the possibilities of our robotics middleware.

We created writing to familiarize ourselves with the API of the system, and later expanded on the application to implement drawing and enable the user to replicate handwriting, signatures, and SVG images.

We started to develop object grasping and manipulation to turn the robotic arm into a polyvalent assistant capable of interacting with its environment and aid in automating manual tasks.

We found that applications fulfilled our criteria for ease of integration and development through this experimentation.

Conclusion 4

In this work, we presented ALFRED as a general purpose robotics middleware, that can be used to control a robotic arm to turn it into an assistant in many domains such as the industry, research, and personal life. We explained the workings of the middleware, using Docker and Docker Compose to create a separation of permissions and responsibilities in the Kernel and the Userspace, centered around a fast and robust message bus enabling communication between components. We presented interactions that were designed with ALFRED, and how they make us think differently about robotic arms by turning them into smart assistants or an extension of ourselves. We demonstrated applications that were created for ALFRED and then deployed in the system, thus augmenting its capabilities.

The next step for ALFRED is to refine it into something that can compete with existing middlewares in the robotics space, to make ALFRED into something that truly and completely answers the needs of people, industries and researchers.

One path towards this goal is to expose it to the public for user testing. We plan to see how users develop interactions and applications, and add more drivers and Kernel capabilities according to their needs.

Another path is to add environment analysis. While applications can use the robotic arm's API to detect collisions and prevent unwanted movements, we want this to be a core feature of the system by making the robotic arm able to see its environment and react to it as part of the Kernel. It will make the platform safer and less complicated to develop on.

ALFRED gives the tools necessary to create environments where robotic arms live alongside humans not as replacements, but as partners that increase productivity

and quality of life.