

Computer Science and Mathematics
COD290
B922435

**PROJECT TITLE THAT COULD
GO OVER THREE
LINES**

by Jacob Toller

Supervisor: Dr. H. Cai

Department of Computer Science
Loughborough University

June 2023

Abstract

abstract goes here

Contents

1	Introduction	3
2	My Background	3
3	Hardware Review	3
3.1	Nvidia Jetson Nano	3
3.2	Loughborough University's In-House Lab Robots & Modifications	4
3.3	Personal Laptop	6
4	Literature Review	6
4.1	6DRepNet	6
4.2	MTCNN	8
4.3	OpenAI & ChatGPT	8
5	Gap Analysis	11
6	Methodology	11
6.1	My Design & Development Strategy	11
6.2	Language & Framework Choice	13
6.2.1	Robot Operating System (ROS2)	13
6.2.2	Python	15
6.3	Software & Supporting Systems	15
6.3.1	VSCode	15
6.3.2	GitHub	15
6.3.3	Ubuntu	16
6.3.4	L ^A T _E X & Overleaf	16
7	System Development	16
7.1	Preliminary Testing	16
7.2	System Requirements	16
7.3	Developing the Attention Sub-System	17
7.4	Developing the Conversation Sub-System	17

7.4.1	Sub-System Requirements	17
7.4.2	Prototyping for response generation	17
7.4.3	Sub-System Design	22
7.4.4	Sub-System Implementation	24
7.4.5	Sub-System Testing & Demonstration	24
8	Appendix	30

Disclaimer

Note that throughout this report, I may refer to 'ROS2' simply as 'ROS'. Robot Operating System 1 (ROS1) has no relevance in this project or report.

All images and diagrams in this document are my own; unless specified otherwise.

1 Introduction

2 My Background

3 Hardware Review

3.1 Nvidia Jetson Nano

Sitting at the heart of this project is the Jetson Nano Developer Kit. It is an ARM-architecture, low-power microcomputer originally released by NVIDIA in 2019 (NVIDIA Newsroom 2019). To disambiguate; the Jetson Nano itself is the smaller of the two PCBs, with the heatsink attached to it, and containing the CPU and GPU. The larger 'motherboard' is what collectively forms the Jetson Nano Developer Kit (JN-DK); and contains IO ports, the WiFi adapter, and the GPIO¹ pins.

Compared to its competitors; such as the beloved Raspberry Pi and its derivatives or the Arduino family of products; the JN-DK is suited to my project as it has the following advantages:

- Enough processing power and memory to support the additional overhead of a 'normal', non-specialised operating system.

¹General-purpose input/output



Figure 1: Image of the NVIDIA Jetson Nano Developer Kit.

- Dedicated GPU. The utilisation of a dedicated GPU is important for applying machine-learning based models; as it allows for significantly faster processing than what would otherwise be possible on just a CPU, especially that of an ARM-based single-board computer. 128 processing cores
- On-board WiFi chip & antenna. This allows for wireless connectivity to the robot.

However, the Jetson Nano does not come without its downsides; which have to be worked around:

- Due to lack of support from NVIDIA regarding the Nano itself; the latest version of CUDA that can be used is 10.2; and must be used in a Ubuntu 18.04 environment (NVIDIA Developer 2022).
- ARM-based CPU architecture means that we have to source libraries & software that have either been distributed with an ARM build; or build one from the source code.

3.2 Loughborough University's In-House Lab Robots & Modifications

If the Jetson Nano is the heart of this project; then the Lab Robots are the body of it.

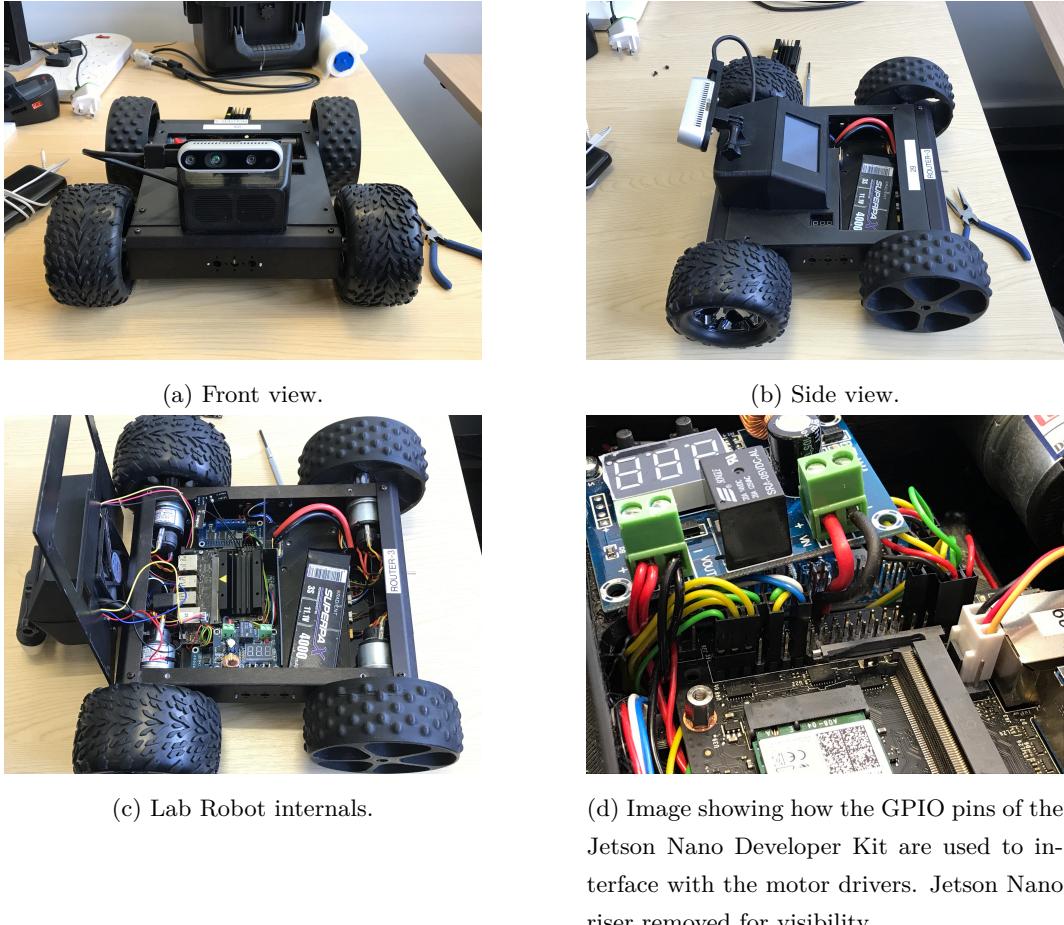


Figure 2: Images of the Lab Robot and its components.

As pictured in Figure ; the Lab Robots use a four-wheeled configuration, where all four wheels are driven by individually addressable, geared, reversible motors. This allows the robot to easily drive forwards and backwards; and can steer using a skid-steer configuration². The robot is powered by a 11.1V, 4000mAh battery; which is used to power both the Jetson Nano and the two motor drivers. This is all enclosed in an aluminium and acrylic casing, which protects the internal components. Atop the body of the robot is a raised compartment which houses a fan to provide airflow to the enclosure; and a mounting for the Intel RealSense D435i camera system. The fan is important as while the passive heat sink of the Jetson Nano is enough to cool the board when exposed; it needs forced airflow when enclosed.

The Intel RealSense D435i is an all-in-one sensing module; that has the following features

²For example; for the robot to turn left, the wheels on the right hand side of the robot can be driven forwards, and the wheels on the left hand side driven backwards.

(Intel RealSense 2019):

- Connectivity via USB-C
- Standard RGB camera; similar to that of a normal webcam. It is able to produce a maximum image size of 1920x1080, at a maximum rate of 30 frames per second.
- Depth sensing; via a stereoscopic pair of cameras. It is able to produce a depth image with a 1280x720 resolution at up to 90 fps.
- Internal IMU (Inertial Measurement Unit) to allow for translational and rotational movement sensing.

Most of these features will go unused in this project; as we will mostly be using the RealSense for the fact that it is what is supplied with the Lab Robots; although they do offer nice opportunities that could be explored in the future. Let us now explore what the robot doesn't have:

- For the robot to be able to perform speech recognition, I needed to add a microphone to the robot.
- In order for the robot to then read out a generated response, the robot also needed a speaker.

3.3 Personal Laptop

Also of note for this project is my personal laptop. In addition to testing the parts of this project as they were developed; was a **FINISH THIS**.

For its purpose here; the following specs are relevant:

- Intel i5-8300H
- NVIDIA GTX 1050, offering 640 CUDA cores.

4 Literature Review

4.1 6DRepNet

One of the sub-systems that I needed for this project was to have a way of determining whether or not a person was looking at the robot. The idea being that someone who would be looking at the robot is paying attention to it and wants to interact.

In the paper “6d Rotation Representation For Unconstrained Head Pose Estimation”, (Hempel, Abdelrahman, and Al-Hamadi 2022a); the authors propose an unconstrained,

landmark-free approach, '6DRepNet', to the task of head pose estimation; where 'unconstrained' means that it should be able to estimate the direction that a person's head is facing from any reasonable direction, not just a narrow angle at the front. The two main approaches for this task are either landmark-based, or landmark-free:

- A model/approach/implementation that uses a landmark-based approach will operate by taking the image of the head and first attempt to locate a series of landmarks on the face (this could be eyes, nose; but also abstract points on the face). These landmarks are then used to attempt to predict a facing direction by comparing it to a reference frame. The accuracy of this type of model is dependent on accurate localisation of the reference landmarks.
- A landmark-free model will attempt to resolve the facing direction without the intermediate step; estimating the facing direction straight from the input image.

Specifically, how their approach differs from other approaches is that their model predicts direction as represented by two 3D column vectors, a_1 and a_2 ; where the 6 elements that make up these vectors are predicted by the model, and is where the model gets its namesake. These vectors are then converted into a 3D rotation matrix with the following equations:

$$\begin{aligned} b_1 &= \frac{a_1}{\|a_1\|} \\ b_2 &= \frac{u_2}{\|u_2\|}, \quad u_2 = a_2 - (b_1 \dot{a}_2) b_1 \\ b_3 &= b_1 \times b_2 \end{aligned} \tag{1}$$

Finalised rotation matrix as:

$$B = \begin{pmatrix} | & | & | \\ b_1 & b_2 & b_3 \\ | & | & | \end{pmatrix} \tag{2}$$

as taken from Eq's. (2-3); (Hempel, Abdelrahman, and Al-Hamadi 2022a). Whereas a typical head-pose detection approach might just directly predict pitch, yaw and roll angles.

This system is applicable to my project if we make the assumption that 'if someone's head is facing towards the robot, then they must be looking at the robot'. My reasoning for using this method over a gaze detection model is that, in theory, it should offer more accurate prediction at range, or lower camera resolution³; as the head is a much larger target to determine direction from compared to using the direction of the eyeballs.

³This could be due to hardware limitations of the camera, or by processing a lower resolution image for transmission/prediction.

In terms of implementation; this paper comes with an accompanying GitHub repository (Hempel, Abdelrahman, and Al-Hamadi 2022b); containing a PyTorch implementation of the model and its weights. The disadvantage to this approach is that the model is only functional when the input image is cropped to just the face. Hence it must be paired with a method of detecting a face/faces from an input image, to generate the cropped images automatically.

4.2 MTCNN

As established in the previous section; I needed a face detector that could be executed on an image to feed bounding boxes into the head-pose detector.

The face detection model that I chose was MTCNN; as published by (K. Zhang et al. 2016).

MTCNN operates in a three stage process. In between each stage, Non-Maximum Suppression (NMS)⁴ is applied to the generated bounding boxes to remove repeated results.

1. The first stage takes the input image and generates a rough set of candidate boxes.
2. The second stage is used to sample and discard boxes from the first stage; and to improve the quality of the remaining boxes.
3. The third, final stage repeats the process of the 2nd stage to produce a finalised prediction. It also, ironically, detects facial landmarks; although they are not numerous enough to be usable for direct head-pose estimation.

To use in my system; I use the python implementation available from the <https://github.com/timesler/facenet-pytorch> repository, (Esler 2019). This specific implementation of the model is a PyTorch implementation. My exact reasoning for this choice will be discussed in ??.

4.3 OpenAI & ChatGPT

Open AI is an American company which develops a variety of Artificial Intelligence models; the purpose of which can vary from 'Synthesising images from a text prompt' (Ramesh et al. 2021), to 'Defeating professional e-sports players in the game Dota 2' (Brockman et al. 2018). They also perform research into the environment surrounding AI; such as *Preparing for malicious uses of AI* (Clark, Page, and Amodei 2018). This goes to show that they have a diverse portfolio/roster of research in the field of AI and its applications.

⁴This is a pruning technique that is used to remove duplicated bounding boxes; where if the Intersection over Union score exceeds a set threshold, the box with the lower confidence score is discarded.

However by far their most renowned⁵ product is ChatGPT; a Generative Pre-trained Transformer model(Brown et al. 2020) packaged as an online chat application, releasing in November of 2022 (OpenAI 2022). Its appeal comes from the fact that users can provide it with a prompt or question; and it will synthesise a reply to assist/answer the user. This is best demonstrated with an example (Figure 3):

⁵in terms of academia and education, 'infamous' is probably more accurate

USER: What is the appeal of ChatGPT?

CHATGPT: The appeal of ChatGPT, and similar language models, lies in their ability to understand and generate human-like text, making them valuable tools for a wide range of applications. Here are some key aspects that contribute to the appeal of ChatGPT:

- Conversational Interaction: ChatGPT is designed to engage in natural language conversations. Users can ask questions, seek information, discuss topics, or have interactive exchanges. The model's ability to understand context and generate relevant responses makes it feel more like conversing with a human.
- Versatility: ChatGPT can handle various topics and domains, drawing upon its training on a diverse range of internet text. It can discuss science, technology, literature, entertainment, history, and more. The broad knowledge base allows for exploration of different subjects within a single platform.
- Information Retrieval: Users can obtain quick and concise information from ChatGPT. It can provide definitions, explanations, summaries, or factual details. This feature is particularly useful for those seeking answers or clarifications on specific topics.
- Creative Writing: ChatGPT can assist with generating text creatively. Whether it's brainstorming ideas, writing stories, or composing poems, ChatGPT can offer suggestions, help overcome writer's block, and act as a collaborative writing partner.
- Learning and Education: ChatGPT can serve as an educational resource, helping users grasp complex concepts or explaining topics in a simplified manner. It can also provide examples, references, or additional resources for further exploration.
- Accessibility: ChatGPT is accessible through various platforms and interfaces, such as web applications, APIs, or integrations with other software. This availability enables users to engage with the model using familiar tools and frameworks.
- Continuous Improvement: ChatGPT is a part of an ongoing research and development process. Feedback from users helps in refining the model and addressing its limitations. This iterative approach ensures that ChatGPT evolves and improves over time.

It's important to note that while ChatGPT can produce human-like responses, it's still an AI model and may occasionally provide incorrect or nonsensical answers. Care should be taken to verify information obtained from any AI-based system.

Figure 3: Example dialogue with ChatGPT. Fresh conversation instance (no history), queried 27/05/2023.

The concept of a Generative Pre-trained Transformer is that it is a model capable of generating a series of tokens⁶ from a set of input tokens; where the 'pre-trained' aspect comes from the fact that these models are trained on an expansive data-set of published media. This includes books, papers and 'formal' publications; but also other sources such as personal websites and social media.

This type of model is applicable to my project, as in concept, we could have the robot listen to what an interactee is saying to it; parse this through a GPT model to generate a response; and then have the robot speak that response aloud for the person to hear.

Regarding actually implementing this; the pre-trained model cannot be locally downloaded and used; due to its immense size and the fact that it is not available for download. Instead, OpenAI offer an API (and accompanying Python wrapper) which can be used to interact and generate results from their various models; including sending prompts to their GPT models for it to generate a reply.

5 Gap Analysis

6 Methodology

6.1 My Design & Development Strategy

In terms of my design and development strategy; I started the project with preliminary testing on some of the libraries/technologies that I could use for my system. This also included the 'infrastructure' of the project, such as how my code would communicate between two different devices; this is where the decision to use ROS2 came from.

Once I had an idea of what I wanted to use and what features I wanted to create; I would build a set of requirements for the component/sub-system that could be used to develop and eventually evaluate it against. To develop, I would either jump to implementing my existing code into ROS; or for more complex parts I would continue working on refining a prototype before moving to ROS.

Once a sub-system was complete, it would be tested in isolation against its own requirements; and then finally integrated into the full system.

⁶These tokens are either words, or parts of words; and is how a block of text is split up into components to be used as an input and produced as output.

Throughout the project, I would have a few formal meetings scattered across the whole year; used to establish the initial direction of the project, and to re-establish the direction of the project if I felt lost. These would be used to supplement the fairly frequent informal meetings that I would also have whenever my supervisor and I happened to be in the same room.

Towards the end of the project, once most of the other deadlines were out of the way; I switched to an Agile-esque workflow; where I would keep track of the remaining tasks that had to be completed on project and the report via a physical (simplified) Kanban board.

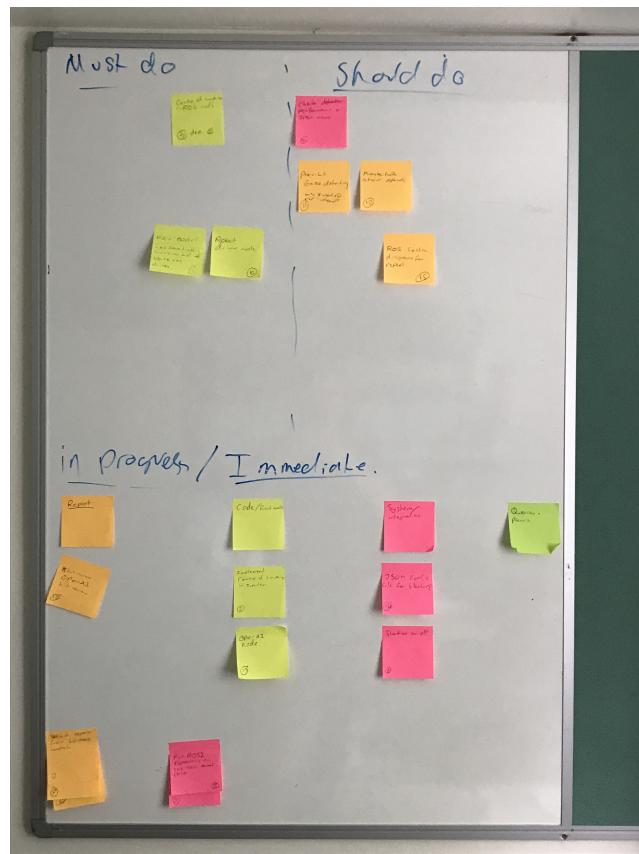


Figure 4: Image of the simplified Kanban board I used to track which tasks needed to be done for the end of the project.

6.2 Language & Framework Choice

6.2.1 Robot Operating System (ROS2)

One of the things that I did at the start of this project was to establish a series of requirements that the backbone of the system would have to fulfil to allow for this project to succeed:

1. I needed a solution that could allow for code to be executed simultaneously. The reasoning for this is simple, as going into the project I recognised that it would have several components that need to operate at the same time; such as face recognition while also maintaining a conversation.
2. I wanted a solution that allowed for the code running on the Lab Robot to easily communicate with another computer. This is because I wanted a backup plan incase something didn't perform well enough/couldn't be run on the Jetson Nano. This would also open up future possibilities for robots which don't have as much processing power available to them locally.
3. I wanted something that could be modular and adaptable. For example could have a centralised control system, that isn't robot specific; and then have a 'client' or something that interfaces with the robot and its specific components. This would mean that the whole project wouldn't have to be 'ported' to a different type of robot, just the inter-facial component.

I chose to use ROS2 for this project as it offers a solution to each of the problems. Robot Operating System 2 (ROS2), as published by (Macenski et al. 2022), is an open-source robotics framework built from the successes and shortcomings of ROS1; and operates on a system of nodes; where each individual node is its own program, written in any language of the developer's choice⁷. These nodes can then communicate between each-other through several different communication types:

- Topics; where Nodes can publish a message to the topic (publisher Nodes); and the message will be distributed to all the subscriber nodes of that topic.
- Services; where server nodes can lie in wait until a client node sends a request to perform a task. Once completed, the server node will send a reply in the form of a response message.
- Actions; which are intended for tasks which take a long time to complete and allow for progress to be reported as it progressed. Functionally, they operated like a combination of a Service and a Topic, with the latter used to publish task feedback.

⁷By default, ROS2 comes with ROS Client Libraries for C, C++ and Python; but community RCL's allow for nodes to be written in other languages.

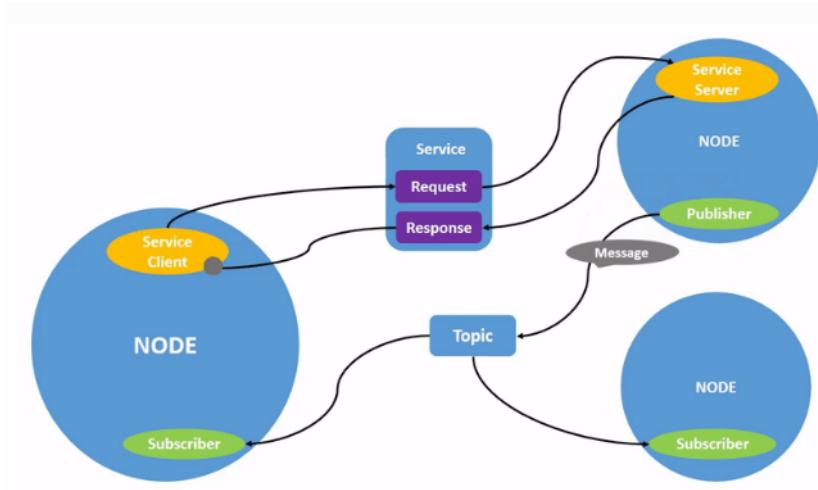


Figure 5: Diagram comparing the functionality of Services and Topics in ROS. Image from (Open Robotics 2021).

It is noted that a singular Node can make use of multiple different communication types, or even multiple of the same type. For example, Figure 5 shows how a possible node could act both as a publisher for a topic, and as a server for a service. This is well utilised in my project.

The node system of ROS solves two of the problems specified earlier; namely allowing for simultaneous processing and modularity. This is because each node functions as its own individual program, which allows for the Operating System of the computer it is deployed upon to manage the allocation of processing time and cores for the process. Having multiple nodes means we can establish the system across a series of 'core' nodes, and then have a node (or even multiple nodes) which is written specifically for the robot it is deployed on, which acts as the interface between the rest of the ROS system and the robot's non-standardised components.

The final issue that ROS solves is that of allowing communication between multiple devices. ROS solves this by allowing for nodes to communicate with nodes that are running on a different device on the same network; as if they were both operating on the same device. Seamlessly. This is big as it not only allows for parts of the system to perform their processing off of the robot itself; but also allows for nodes to be tested on the main system as they are developed; as the system could be running entirely on the Jetson of the robot, with the exception of the in-development node being tested and iterated from my laptop. The requirements for this to function is simply having the devices on the same network, and

having the same `ROS_DOMAIN_ID`⁸.

On the Jetson Nano, ROS2 version Dashing Diademata is used; while on my laptop, 'Galactic Geochelone' is used. Different versions correspond to the different versions of the operating system used; but this has no impact on the project whatsoever.

6.2.2 Python

My choice of programming language for this project was Python 3; due to my familiarity with it and its popularity meaning that there are plenty of useful libraries to choose from. The non-standard libraries that I used will be mentioned when discussing the development of the individual subsystems.

To augment Python and make the testing/prototyping stage easier; this was supplemented with Jupyter Notebooks; which (for my use-case) provides an alternative way of executing python code in the form of separated blocks of code, and with widgets that make displaying information a lot easier.

6.3 Software & Supporting Systems

6.3.1 VSCode

VSCode is my text-editor/IDE of choice, regardless of project. Its suitable here as it supports all of the filetypes of my project, with syntax highlighting; notably .py, .json, .cfg and .xml files.

6.3.2 GitHub

Git & GitHub were used extensively in order to manage the code-base of this project. I elected to manage the project across two separate repositories; with one repository used for holding all of my prototyping work and report-writing materials, and the other used to hold the collection of ROS packages. My reasoning for this is that it allowed me to keep one of the repositories purely ROS-based so that it could easily be deployed onto the Jetson Board from development on my laptop.

[`https://github.com/plebble/human-robot-interaction`](https://github.com/plebble/human-robot-interaction)
(For prototypes and non-ROS code.)

[`https://github.com/plebble/ros2-project`](https://github.com/plebble/ros2-project)
(ROS packages for deployment on the robot.)

⁸This is used to allow for two separate ROS systems to operate on the same device/devices/network without interference.

6.3.3 Ubuntu

I chose to use Ubuntu when developing this system as it meant that I could develop & test on the same operating system that the Jetson Nano would be using (albeit on a different version; 20.04 vs an ARM build of 18.04). Dual-booting this with windows meant that I could strictly keep work on Ubuntu to improve workflow as well.

6.3.4 L^AT_EX & Overleaf

In order to write and typeset this thesis; I am making use of L^AT_EX and the Overleaf editor/platform. I've worked with a local installation of the L^AT_EX environment in the past, on Windows and it was one of the most painful installation experiences in my life. Overleaf bypasses this as it is a fully comprehensive editor and compiler accessible from a browser.

7 System Development

7.1 Preliminary Testing

Preliminary testing started with experimenting with the python `SpeechRecognition` library (A. Zhang 2023) and `pyttsx3` (Bhat 2020); with the idea that the robot would eventually be able to converse with the interactee. The testing here was a success in terms of verifying technical ability; but in terms of code that could be used for the main project, only resulted in a few fine tuning parameters for the speech recognition. Note that at this point in the project; ChatGPT had just become available for public use, but the API to programmatically interface with it had not yet been released.

It was in this stage where I also looked into testing the `sixdrepnet` and `facenet-pytorch` libraries that were mentioned in 4. This was to ensure that I could get real-time performance out of them; as without this, they could not be used to interact with a person without an uncomfortable delay. This was successful, as I was able to get detection speeds that were more than required for it to work.

7.2 System Requirements

After preliminary testing, I came up with a set of requirements that the system as a whole would have to fulfil, in order to complete the project:

- The system should be able to figure out if a person wants to interact with the robot
- Once it has correctly discerned that a person wants to interact; listen to what they say and

- The system should be able to function both on and off-board of the robot. This is to facilitate future deployments/usages on other physical robots which may not have the processing strength of the Jetson Nano.
- The system's components & sub-systems should be modular in nature; such that a component can be disabled entirely if we do not want its functionality, or replaced seamlessly with a component that does the same task with a different method. This could be something such as a different method of face detection, or replacing the Text-to-Speech node with a more realistic sounding speech-synthesis model; and provides opportunity for future upgrades.

The first two points correspond to individual sub-systems of the system; and each will have their own set of requirements that were used to build them.

7.3 Developing the Attention Sub-System

7.4 Developing the Conversation Sub-System

7.4.1 Sub-System Requirements

This subsystem relates to what the robot will do when it has someone that it wants to interact with by conversing. At the point when these requirements were written; the API for the model that ChatGPT uses (`gpt-3.5-turbo`) had become available; meaning that there was some feasibility to these requirements:

- When interacting with someone, the sub-system should be able to use a form of speech recognition in order to transcribe what the user is saying into text
- The sub-system should then use a GPT model in order to generate a response. The tone of the response could either be like a helpful assistant, or informal like talking to a friend.
- The sub-system needs to take the response, and then somehow communicate it back to the user; for example through text to speech.
- The sub-system should also take care as to not pickup its own audio when using TTS; as this could form a feedback loop where the robot is constantly replying to itself.

7.4.2 Prototyping for response generation

I started this by experimenting with the `openai` API wrapper, and the `gpt-3.5-turbo` model. This model is the one that is used to generate responses for the current public

release version of ChatGPT. To use this, we have to generate a `system` message, which acts as context.

```
system_msg = {"role": "system", "content": "<context message goes here>"}
```

I would use the context message "You are to play the role of a small robot designed for human-robot interaction. Please answer with short responses always." for this to tell the model what it is being used for when it generates its responses. After, the messages of the user are packaged in a similar format:

```
user_msg = {"role": "user", "content": "<user message goes here>"}
```

The content of this message would be equivalent to sending a message in the chat box of ChatGPT; and could be a question, a greeting, a query etc. These two messages are then packaged together as a list, and then it is sent as an API request to OpenAI.

```
msg_list = [system_msg, user_msg]
response = openai.ChatCompletion.create(model="gpt-3.5-turbo", messages=msg_list)
```

With the user message as "hello robot, how are you?"; the request yields the following response, in standard JSON format.

```
{
  "id": "chatcmpl-7LUH5fZ0R1cZDd0x8eW6xEMWHlRgj",
  "object": "chat.completion",
  "created": 1685355619,
  "model": "gpt-3.5-turbo-0301",
  "usage": {
    "prompt_tokens": 44,
    "completion_tokens": 8,
    "total_tokens": 52
  },
  "choices": [
    {
      "message": {
        "role": "assistant",
        "content": "I am functioning well, thank you."
      },
      "finish_reason": "stop",
      "index": 0
    }
  ]
}
```

This gives us high quality responses that we can extract the generated message out of and send to the Text-to-Speech node to be read out to the user. However, this comes with disadvantages. In order to use the OpenAI API, an API key is mandatory; and can be obtained for free by registering with them. On a free plan with OpenAI, the rate limit is 3 Requests Per Minute for the `gpt-3.5-turbo` model (see Figure 6).

Rate limits

OpenAI enforces rate limits on the requests you can make to the API. These are applied over requests-per-minute, tokens-per-minute, or in the case of image models, images-per-minute.

Learn more in the [rate limits documentation](#), or reference the [default rate limits](#) for our models. Your organization's rate limits are listed below.

MODEL	RPM	TPM
CHAT		
gpt-3.5-turbo	3	40,000
gpt-3.5-turbo-0301	3	40,000
TEXT		
ada	60	150,000
ada-code-search-code	60	150,000
ada-code-search-text	60	150,000
ada-search-document	60	150,000
ada-search-query	60	150,000
ada-similarity	60	150,000
babbage	60	150,000
curie	60	150,000
text-davinci-002	60	150,000
text-davinci-003	60	150,000

Figure 6: Screenshot of OpenAI's API limits for a free account; from (OpenAI 2022).

This led me to explore what was possible with the models that were specified for text purposes, instead of a model that has been specifically trained for use as a chat assistant. These models are designed for the task of 'Text Completion': to examine the piece of text it has been provided with, and then generate a series of tokens (words) that fit in on the end of the prompt. For example, when prompted with a few opening lines from *Of mice and men* (Steinbeck 2023)

```
text = """A FEW MILES south of Soledad, the Salinas River drops in close
to the hillside bank and runs deep and green. The water is warm
too, for it has slipped twinkling over the yellow sands in the
sunlight before reaching the narrow pool."""
```

```
response = openai.Completion.create(model = "text-davinci-003",
prompt=text,n=1,max_tokens=16)
```

it is able to understand that it is being provided with a piece of existing material and it can follow on by accurately reciting the text that follows:

```
{
  "id": "cmpl-7M00Hes0Y0wU7vIqalUz95CTDf0eq",
  "object": "text_completion",
  "created": 1685477585,
  "model": "text-davinci-003",
  "choices": [
    {
      "text": "On one side of the river the golden foothill slopes curve up to",
      "index": 0,
      "logprobs": null,
      "finish_reason": "length"
    }
  ],
  "usage": {
    "prompt_tokens": 59,
    "completion_tokens": 16,
    "total_tokens": 75
  }
}
```

Not only can it be used to complete the text of existing works, but it is also able to complete shorter, more abstract prompts; where it has to specifically come up with new content based on the facts, instead of quoting something. For example,

```
description = """Describing a car:
```

```
A car has"""
```

gets the API response of:

```
...
{
  "text": " four wheels, a steering wheel, and a motor. It is designed to transport",
  "index": 0,
  "logprobs": null,
  "finish_reason": "length"
}
...
```

Note that the end of these responses is cut off due to "length". This is because we specified that the reply should have a max token count of 16. The upsides to using the `text-davinci-003` model instead of the ChatGPT model is that we are not as restricted with the amount of requests that we can make, or the number of tokens we can use⁹; as we can send up to 60 requests per minute which is more than enough for our needs. We can 'trick' this model into becoming a chat model by formatting our prompt as dialogue between two 'characters':

```
dialogue = """
Context: The SYSTEM is a small robot designed for human-robot interaction;
and replies with short answers.
```

```
DIALOG START
USER: How are you today?
SYSTEM: """
```

```
response = openai.Completion.create(model = "text-davinci-003",
prompt=dialogue,n=1,max_tokens=320)
```

this yields the response

```
{
  "id": "cmpl-7M0qJfDSYAtMFkRTq4dDhYjYvzpq2",
  "object": "text_completion",
  "created": 1685480811,
  "model": "text-davinci-003",
  "choices": [
    {
      "text": " Hi, I'm doing great! How can I help you?",
      "index": 0,
      "logprobs": null,
      "finish_reason": "stop"
    }
  ],
  "usage": {
    "prompt_tokens": 45,
    "completion_tokens": 13,
    "total_tokens": 58
  }
}
```

⁹For our uses, the RPM is the limiting factor, not the token count (1000 tokens \approx 750 words(OpenAI 2023)). The token count is the total between the request and the reply.

If we tell it that the 'SYSTEM' character replies with short answers; we can raise the number of tokens it is allowed to reply with and it will instead limit itself so that what it produces isn't cut off in any way. We can keep a log of what is said for continued discussion instead of one-off answers; by adding the model's reply to the end of the dialogue as 'SYSTEM'. Whenever the interactee says anything, we add it to the dialogue as 'USER: <recognised text>; then add a blank message, 'SYSTEM: ', so that the model knows it has to complete the reply of the 'SYSTEM' character and act as instructed at the start.

7.4.3 Sub-System Design

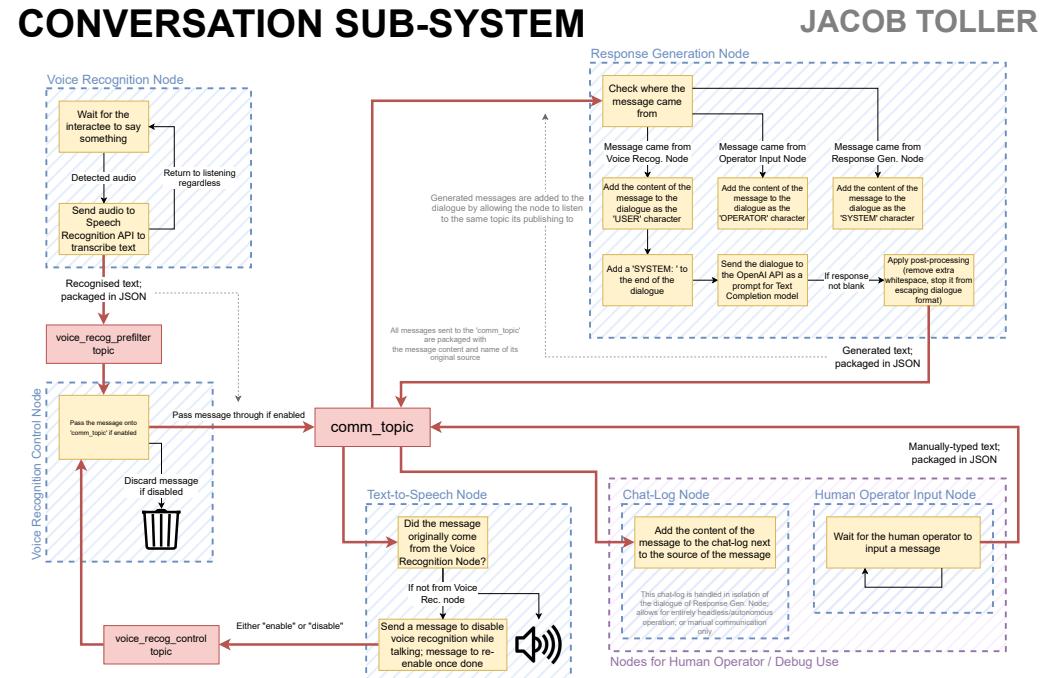


Figure 7: Flowchart showing the individual processes of each node; and how they communicate between each-other.

In this flowchart (Figure 7); the process generally starts at the Voice Recognition node, when the interactee is talking. The node will wait until it hears something that could be speech, and once the speech has stopped, the recorded audio is processed to transcribe text. If no text is recognised, just start listening again; otherwise package it alongside its context into JSON format, and broadcast it on the `voice_recog_prefilter` topic of our ROS environment.

```
{  
    "listening_start": 1685525716,  
    "timestamp": 1685525719,  
    "source": "voice_recog",  
    "text": "hello how are you"  
}
```

The message has 3 standard attributes which are common for every message that is published to the `comm_topic`. The "`source`" attribute always takes the value of the internal name of the node it originally came from, and is used in decision making down the line; with the "`text`" attribute used for the content of the message, and "`timestamp`" tracking the Unix timestamp that the message was sent at. Specific to the Voice Recognition node is "`listening start`", the timestamp that the microphone started listening for to register that text.

The only node that is subscribed to the `voice_recog_prefilter` topic is the Voice Rec. Control node; and its only purpose is to decide whether or not a message from the Voice Recognition node should be allowed to pass. In terms of design; the reason that this is required is that it is what stops anything that the robot says from immediately being recognised by the Voice Recognition node and forming a feedback loop of the robot constantly answering to itself. This node has an internal flag as to whether it should allow the messages through or not; if allowed, the message is forwarded to `comm_topic` without modification.

There are three unique nodes that are subscribed and waiting for messages on `comm_topic`. The most important is the Response Generation node, which receives and acts of all messages. It starts by checking where the message came from, and adding the message to its internal dialogue string with the appropriate label, as per 7.4.2 and Fig. 7. If the message was flagged as a 'USER' message, we assume that the interactee is expecting the robot to generate a reply, so we follow the procedure outlined in the prototyping stage to generate a response. This response is then wrapped up in the standard format (as per the example above, except without the "`listening start`" attribute) and published to `comm_topic`. Note how we didn't add the generated reply to the dialogue, this is because we allow this node to listen for *any* message, but only reply to ones from the user.

The other main node that is subscribed to `comm_topic` is the Text-to-Speech node. This one also starts by checking the source of the message, and as long as it didn't originate from the Voice Recognition node; it is send to the TTS engine for dictation. When the node starts speaking, a simple message is published to the `voice_recog_control` topic to tell the Voice Rec. Control node to disable future speech from being recognised; and another message is published once the node has stopped speaking to re-enable voice recognition.

The other two nodes in this system are for the use of a human operator. The Chat-Log node simply receives messages and prints them to the terminal for the operator to read; and the Operator Input Node simply allows the operator to send messages from their terminal for the robot to read-aloud. These nodes don't have to be active for this subsystem to be functional.

7.4.4 Sub-System Implementation

While the previous section should give a good overview of how this sub-system is designed and structured; to the point where it should be easily reproducible by someone attempting to follow in my footsteps; I will now follow up with miscellaneous points regarding my specific implementation.

- The Voice Recognition node must be located on the robot, as it directly uses the microphone and recognises the speech within the same node. For future implementations/versions, it may be advantageous to have the audio recording and then speech detection on two separate nodes allowing them to be on different devices.
- Due to the GPT model that I used only being accessible through an API; the device that the Response Generation node is running on needs internet access. For usage, the OpenAI API key should be stored as an environment variable as `OPENAI_KEY`
- Similarly, the Voice Recognition node needs internet access when using the Google Speech Recognition API for detection. The `SpeechRecognition` library has the capability to use a variety of different providers, including several local, offline ones; but the best results were obtained from using the Google API.
- The Voice Recognition Control node specifically checks whether the message came while the 'is enabled?' flag was false **or** if the audio recording that it came from precedes the last time the control flag was set to enabled.
- After implementation; I found that sometimes the response generated by the GPT model would try to 'break free' from the format of the dialogue; sometimes inserting an extra "SYSTEM:" line or finishing its answer then continuing to write a reply for the USER. The post processing makes sure that any attempts at doing this are pruned so that it is just the message its supposed to make.

7.4.5 Sub-System Testing & Demonstration

I will now demonstrate/showcase this system.

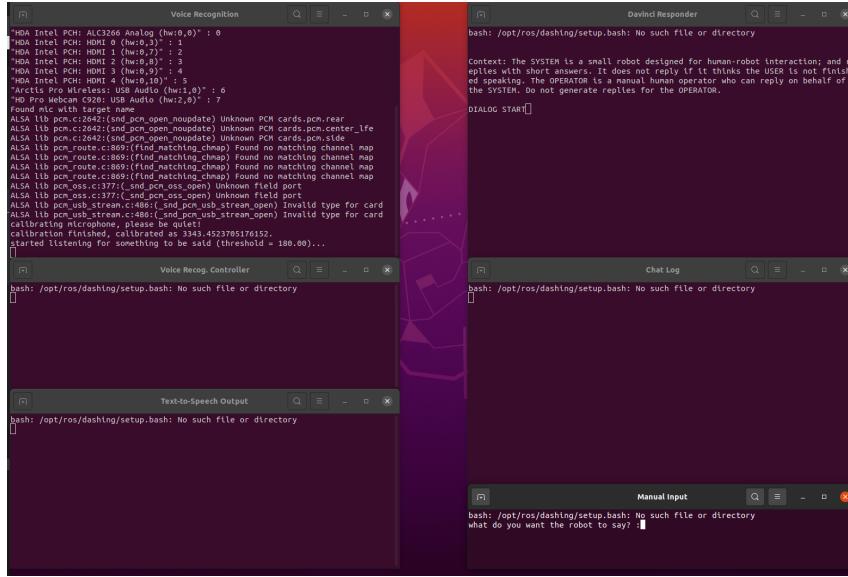


Figure 8: Screenshot of each node in the Communications Sub-System, in their initial state.

Figure 8 shows the state of each of the 6 nodes as they are initialised. This is running entirely on my laptop, which should be analogous to running the system entirely on the Jetson Nano. None of these nodes are currently doing anything, just idly waiting; with the exception of the Voice Recognition node which has started listening for audio. If we then say something, this should be picked up:

```
started listening for something to be said (threshold = 180.00)...
stopped listening, sending audio...
speech recognised as: hello robot how are you
{
  "listening_start": 1685542483.9754453,
  "timestamp": 1685542492.5655754,
  "source": "voice_recog",
  "text": "hello robot how are you"
}
started listening for something to be said (threshold = 180.00)...
```

- (a) Once it thinks the talking has stopped, it is send for transcription. The encoded message is then published and the Voice Recognition node starts listening again.

```
jacob@JACOB-XPS-UBUNTU:~/ros2_ws$ ros2 run py_communication davinci_responder

Context: THE SYSTEM is a small robot designed for human-robot interaction; and r
eplies with short answers. It does not reply if it thinks the USER is not finish
ed speaking. THE OPERATOR is a manual human operator who can reply on behalf of
the SYSTEM. Do not generate replies for the OPERATOR.

DIALOG START

USER: hello robot how are you
SYSTEM: Hi there! I'm doing well. How about you?[]
```

- (c) The Response Generation node adds the user's text to the dialogue, generates & sends the result, then immediately receives its own message and adds it the dialogue; as displayed here.

```
jacob@JACOB-XPS-UBUNTU:~/ros2_ws$ ros2 run py_communication voice_recog_control
Enabled voice recog.? : True
Received message from 'voice_recog_prefilter', parsing JSON...
Allowing passthrough...
Received message from 'voice_recog_prefilter', parsing JSON...
Allowing passthrough...
```

- (b) Voice recognition is currently enabled, so the message is passed through.

```
"listening_start": 1685542483.9754453,
"timestamp": 1685542492.5655754,
"source": "voice_recog",
"text": "hello robot how are you"
}
Received voice_recognition; not speaking because it is me.
Received message from 'comm_topic', parsing JSON...
{
  "timestamp": 1685542493.685951,
  "source": "tts_speech_node",
  "text": "Hi there! I'm doing well. How about you?"
}
Sending message to disable voice recognition, so it doesn't pick up the tts.
Using TTS to say: Hi there! I'm doing well. How about you?
duration to receive and talk: 2.67
[]
```

- (d) Both the recognised speech and the generated response are picked up by the Text-to-Speech node. The former is discarded as the node recognised that its come from the user, we don't want to parrot their words back to them. The second message is spoke aloud and the control messages are sent to the Voice Recog. Control node.

```
jacob@JACOB-XPS-UBUNTU:~/ros2_ws$ ros2 run py_communication voice_recog_control
Enabled voice recog.? : True
Received message from 'voice_recog_prefilter', parsing JSON...
Allowing passthrough...
Received message to disable recognition...
Received message to re-enable recognition...
[]
```

- (e) We can see these control messages reflected in the control node.

```
jacob@JACOB-XPS-UBUNTU:~/ros2_ws$ ros2 topic echo /comm_topic
data: {"\n  \"listening_start\": 1685543709.737897,\n  \"source\": \"voice_recog\", \n  \"text\": \"Hello...\"\n}\n\n
data: {"\n  \"timestamp\": 1685543713.8423445,\n  \"source\": \"tts_speech_node\", \n  \"text\": \"Hi there! I'm doing well. How about you?\"\n}\n\n
data: {"\n  \"timestamp\": 1685543714.3463378,\n  \"source\": \"response_generator\", \n  \"text\": \"Hi there! I'm doing well. How about you?\"\n}\n\n
[...]
```

- (f) This is a screenshot of one of ROS2's command-line tools; that prints the messages of a specified topic (`comm_topic`) as they are received; showing how the messages are encoded in JSON format.

Figure 9: Processing pipeline of the Communication Sub-System as a piece of text is recognised.

In order to demonstrate the last of the requirements; we can use the Manual Operator Input node to send a message that will take a long time for the Text-to-Speech node to read out; then try saying something for Voice Recognition to pick up:

(a) Manual Entry used to send long message to TTS.

```
started listening for something to be said (threshold = 180.00)...
stopped listening, sending audio...
speech recognised as: hello there
{
  "listening_start": 1685550663.8119864,
  "timestamp": 1685550673.696468,
  "source": "voice recog",
  "text": "hello there"
}
started listening for something to be said (threshold = 180.00)...
```

(c) If we now say something for the Voice Recognition node to listen for...

```
started listening for something to be said (threshold = 180.00)...
stopped listening, sending audio...
speech recognised as: are you listening now
{
  "listening_start": 1685550683.176047,
  "timestamp": 1685550688.3125763,
  "source": "voice_recog",
  "text": "are you listening now"
}
started listening for something to be said (threshold = 180.00)...
```

(e) Once the TTS node has finished speaking, it will send another control message to re-enable.

Now if we speak into the microphone...

(b) This is received by the TTS node, it sends a control message to disable while speaking.

```
jacob@JACOB-XPS-UBUNTU:~/ros2_ws$ ros2 run py_communication voice_recog_control
Reclved Message to disable recognition...
Enabled voice recog.? : False
Voice rec. currently disabled; discarding...
```

(d) ...it is blocked by the Control node, as it previously received a control message to disable passthrough.

```
jacob@JACOB-XPS-UBUNTU:~/ros2_ws$ ros2 run py_communication voice_recog_control
Received message to disable recognition...
Enabled voice recog.? : False
Voice rec. currently disabled; discarding...
Received message to re-enable recognition...
Enabled voice recog.? : True
Received message from 'voice_recog_prefilter', parsing JSON...
Allowing passthrough...
[]
```

(f) ...it is allowed through as the control node has since received the message to re-enable.

```
Received message from 'comm_topic', parsing JSON...
{
    "listening_start": 1685580683.176047,
    "timestamp": 1685580688.3125763,
    "source": "voice_recog",
    "text": "are you listening now"
}
Received voice_recognition; not speaking because it is me.
```

(g) This is shown in the TTS node, as it receives the recognised text once published to `comm_topic`; even though it knows not to read it aloud.

Figure 10: Demonstration of how the voice recognition is disabled while the robot is talking via Text-to-Speech

References

- Bhat, Natesh M (July 2020). *pyttsx3*. URL: <https://pypi.org/project/pyttsx3/> (visited on 05/30/2023).
- Brockman, Greg et al. (June 2018). *OpenAI Five*. URL: <https://openai.com/research/openai-five> (visited on 05/28/2023).
- Brown, Tom B. et al. (May 2020). *Language Models are Few-Shot Learners*. arXiv: 2005.14165 [cs.CL].
- Clark, Jack, Michael Page, and Dario Amodei (Feb. 2018). *Preparing for malicious uses of AI*. URL: <https://openai.com/research/preparing-for-malicious-uses-of-ai> (visited on 05/28/2023).
- Esler, Tim (May 2019). *facenet-pytorch*. URL: <https://github.com/timesler/facenet-pytorch> (visited on 05/27/2023).
- Hempel, Thorsten, Ahmed A. Abdelrahman, and Ayoub Al-Hamadi (Oct. 2022a). “6d Rotation Representation For Unconstrained Head Pose Estimation”. In: *2022 IEEE International Conference on Image Processing (ICIP)*. IEEE. DOI: 10.1109/icip46576.2022.9897219. URL: <https://doi.org/10.1109%5C2Ficip46576.2022.9897219> (visited on 05/27/2023).
- (Feb. 2022b). *6DRepNet*. URL: <https://github.com/thohemp/6DRepNet> (visited on 05/27/2023).
- Intel RealSense (Feb. 2019). *Intel RealSense Depth Camera D435i*. URL: <https://www.intelrealsense.com/depth-camera-d435i/> (visited on 05/25/2023).
- Macenski, Steven et al. (May 2022). “Robot Operating System 2: Design, architecture, and uses in the wild”. In: *Science Robotics* 7.66. DOI: 10.1126/scirobotics.abm6074. URL: <https://doi.org/10.1126%5C2Fscirobotics.abm6074> (visited on 05/25/2023).
- NVIDIA Developer (June 2022). *JetPack SDK 4.6 Release Page*. URL: <https://developer.nvidia.com/embedded/jetpack-sdk-46> (visited on 05/25/2023).
- NVIDIA Newsroom (Mar. 2019). *NVIDIA Announces Jetson Nano: \$99 Tiny, Yet Mighty NVIDIA CUDA-X AI Computer That Runs All AI Models*. URL: <https://nvidianews.nvidia.com/news/nvidia-announces-jetson-nano-99-tiny-yet-mighty-nvidia-cuda-x-ai-computer-that-runs-all-ai-models> (visited on 05/25/2023).
- Open Robotics (May 2021). *Understanding Nodes*. URL: <https://docs.ros.org/en/galactic/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html> (visited on 05/25/2023).
- OpenAI (Nov. 2022). *ChatGPT / OpenAI*. URL: <https://chat.openai.com/> (visited on 05/28/2023).
- (Mar. 2023). *Pricing*. URL: <https://openai.com/pricing> (visited on 05/30/2023).

- Ramesh, Aditya et al. (Jan. 2021). *DALL-E: Creating images from text*. URL: <https://openai.com/research/dall-e> (visited on 05/28/2023).
- Steinbeck, John (2023). *Of mice and men*. Penguin Classics, an imprint of Penguin Books.
- Zhang, Anthony (Mar. 2023). *SpeechRecognition*. URL: <https://pypi.org/project/SpeechRecognition/> (visited on 05/30/2023).
- Zhang, Kaipeng et al. (Oct. 2016). “Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks”. In: *IEEE Signal Processing Letters* 23.10, pp. 1499–1503. DOI: 10.1109/lsp.2016.2603342. URL: <https://doi.org/10.1109%5C%2Flsp.2016.2603342> (visited on 05/27/2023).

8 Appendix