Robot Brain Cluster - Enhanced Computer Vision for Robot DE NIRO

Final Report

Nicolas Tobis, Tolga Dur, Eivinas Buktus, Jiacheng Wang,
Joshua Harris, Harry Butt

Supervisor: Dr. Petar Kormushev

May 20, 2019

Abstract

Physical robots, whether they be for industrial, personal or scientific use are often limited by computational constraints imposed by a range of factors. Our goal in this project was to relax these constraints on Imperial College London's Design Engineering Natural Interaction Robot (DE NIRO) through implementing and building on top of a cluster of computers that provides the robot with opportunities to delegate processing to one of multiple machines within said cluster. This provides DE NIRO with more flexibility to perform multiple, complex skills at the same time, culminating in a demonstration of a game of Hide and Seek using advanced computer vision and speech recognition.

Contents

1 Introduction

1.1 Robot DE NIRO

We have worked on this project with Imperial College London's Robot Intelligence Lab [1], led by Dr. Petar Kormushev [2], with a key goal of extending the lab's original robot DE NIRO [3].

Robot DE NIRO (Design Engineering's Natural Interaction Robot) is a research platform that combines manipulation and locomotion intended to understand more deeply robot-human, robot-object interactions and robot learning. The physical hardware was built in the Design Engineering Laboratory and is based on Rethink Robotics' 6-armed Baxter Robot and a mobile base using a retrofitted electric wheelchair. In addition to Baxter's sensors, DE NIRO has been equipped with a range of sensors to augment its perception abilities including 3D and 2D laser scanners, Amazon Echo Dot for audio, Microsoft Kinect camera for depth perception among many others.

This project's key goal was to extend DE NIRO's pre-existing capabilities (Autonomous Snack Delivery Android in 2017 [5], Project Fezzik in 2018 [6]) through teaching a new complex task with integration on a 'brain' cluster made up of 10 computers, culminating in a live demonstration of the advanced capabilities built on top of the brain cluster.

1.2 Hide and Seek

Given the original goal of extending DE NIRO's capabilities onto a cluster of multiple computers, it was important for us to select an end goal to anchor our project. Since much of the work is currently based on grasping and movement within the lab, amongst our MSc Computing Science peers and other students, we decided on improving the robot's perception and interaction capabilities an effective way to showcase it's newly integrated processing power.

Previous attempts at integrating object and facial recognition were relatively simple compared to the current state of the art, much of which was due to restrictions on computing constraints. To do this, we created a routine based on a classic game of Hide and Seek. In this, DE NIRO will be requested to identify faces and objects through seamless voice led interactions, and will use it's array of cameras to look for requested objects or faces, or alternatively identify anything within its vicinity. If DE NIRO is unable to see a requested item, or perhaps misinterprets a voice command, it is also able to resolve conflicts through the voice interface. Combined with this, DE NIRO has also integrated the ability to rotate and point with its database of instructions meaning it can do basic searches for objects or faces that have been requested but can not necessarily see within its field of view.

This simple yet powerful routine is one that is well known and subtly complex. With the vision for DE NIRO to advance research into creating an all encompassing home support tool, it is important to be able to process information from noisy environments and potentially resolve conflicts on the fly. This report describes in detail our approach to integrating a cluster of computing power to develop this complex routine, including system design, tools, changes and extensions as well as areas that were less successful. We hope that our efforts throughout this project, combined with other work gone into advancing DE NIRO's capabilities will enhance future work and provide the backbone for achieving the original vision of a true support robot.

2 Specification and Extensions

2.1 Initial Specification

Upon beginning the project, the primary goal given to us from Dr. Kormushev was to integrate a set of up to 10 computers into a cluster that DE NIRO can delegate expensive processing tasks to. From this, we were given flexibility to research and agree on a set of skills to introduce to the robot to utilise these added resources.

2.2 End Product Definition

The project aims were broadly split into two. Firstly, it extends DE NIRO's processing abilities through creating a 'brain' that is a software architecture capable of utilising a network of multiple CPUs and GPUs to process raw data more effectively. Secondly, the project implements enhanced computer vision techniques for object detection and face recognition, as well as extending speech detection to create a human-robot interface. The bulk of the project essentially consisted in integrating these subcomponents on the cluster that would enable a demonstration of a version of the game Hide and Seek. We will briefly discuss each subcomponent and the initial task specifications, as well as changes and extensions that were made.

2.2.1 User Interface

Baseline

1. Implementing state of the art speech recognition techniques to allow for seamless interactions between humans and DE NIRO.
2. Further audio output capabilities to allow for improved communication abilities.
3. Integrate new skills with the other nodes on the Robot Brain Cluster and Robot DE NIRO.

Extension

1. Integrate multiple languages in speech recognition.
2. Use on-board display to visualise what DE NIRO is 'seeing' or 'thinking'.

2.2.2 Enhanced Computer Vision - *Object and Facial Recognition*

Baseline

1. Conduct a literature review of the current state of the art pre-trained libraries available for the new skill.
2. Create prototype implementations to test the most promising libraries and identify their applicability to our intended integration and demonstration.
3. Implement the best performing algorithm as a package in Robot Operating System (ROS) [7] on one of the computers in the Robot Brain Cluster.
4. Integrate the new skill with the other nodes on the Robot Brain Cluster and Robot DE NIRO.
5. Integrate 'spoofing detection' ability for DE NIRO to detect attempts to trick the facial recognition system with, for example, printouts of faces on paper.

Extension - *Object Recognition*

1. Extend the object datasets that our object detection algorithm is trained on.
2. Train our algorithm on our own image data for objects particularly relevant to DE NIRO.
3. Implement other types of image analysis relating to objects, such as: Object Character Recognition (OCR), object tracking over multiple frames, approximate depth estimation, and caption generation.

Extension - *Facial Recognition*

1. Extend spoofing detection capabilities to objects and thus create a link between the face recognition and object recognition capability
2. Implement emotion detection
3. Implement pose recognition and integrate it with face recognition to enable DE NIRO to match poses with people.

2.2.3 Brain Cluster

Baseline Set up a cluster of multiple PCs to function as DE NIRO's brain for intensive software procedures and data processing, with the on-board computer being the hub for task delegation.

Extension Develop a framework for multiple master nodes (on-board and in the cluster) and intelligent task delegation to allow for most resource intensive processes to be run on the cluster.

2.2.4 Demonstration Routine

Baseline Develop software to allow DE NIRO to play a game of 'Hide and Seek' in which the robot will be asked questions that are variations of 'Where is this person/object?' and will use a combination of its extended capabilities to identify (if it is valid) the location of the object.

Extension Integrate grasping and/or moving abilities to allow DE NIRO to search and fetch an object to whoever asks the question.

2.3 Extensions and Specification Changes

We largely met most of the baseline specifications and implemented versions of our extensions without significant change. There is a detailed discussion of each sub-component within section 5. Much of this was possible with the demonstration in mind to anchor our efforts and have a high level finishing goal in mind. The separation across components also meant we were able to take advantage of popular programming paradigms implemented within ROS, such as State Machine with Services and Publisher/Subscriber which facilitated rapid prototyping and integration of sub-components.

3 Software Design

3.1 Software Components

With an end goal in mind, we separated into three separate sub teams to work on each distinct software component. This gave us the opportunity to not only to work on areas and develop software that was aligned with our personal interests, but also facilitated the rapid development and prototyping of our demonstration routine. As highlighted in section 5, these were as follows:

1. Object Recognition (Owners: Joshua Harris and Eivinas Butkus)
2. Face Recognition and Learning (Owners: Jiacheng Wang and Tolga Dur)
3. User Interface (Owners: Nicolas Tobis and Harry Butt)

Added on top of these were development of the Environment Model, robot movement and creating the brain cluster. Almost all of these components required completely different skill sets to research and develop and would function in different parts of the demonstration routine. Most importantly, progress across each component was not consistent through the project timeline, thus making the integration efforts even more of a challenge. Integration is, of course, the primary challenge in any robotics projects and something we experienced at almost all stages of development. However, having identified this concern at the early stages, we worked hard on ensuring that our work had the best chance of integrating seamlessly to DE NIRO's existing software and vitally, the hardware components required to execute the demonstration. Much of this was enabled through effective and exhaustive high level system design as well as the many tools that Robot Operating System (ROS) has to support robotic projects of all sizes, combined with necessary changes to the project scope along the way.

3.2 System Design and Overview

As presented in Figure 3.1, we spent significant amounts of time researching core capabilities of ROS and how this would facilitate the integration within our project. We also wanted to ensure that projects following on from us would find software that was well documented and most importantly useful. Much of this was through elegant object oriented design, where we intended to make each of the sub-components as loosely coupled as possible.

Whilst there is a detailed discussion of each sub-component features and development approach within 5, it is worth highlighting two main components in our design that are at the center of our project: the Environment Model and the State Machine. The Environment Model aggregates information from our object and face recognition nodes in a smart and meaningful way. This provides DE NIRO with unprecedented situational awareness. For instance, he can say who has what object while also taking into account recently seen frames. While the Environment Model can be seen as the brains, the State Machine is driving the whole routine. It ensures a consistent flow of control taking the user from one state to another. The Environment Model was entirely conceived and built by us, but the raw skeleton for a state machine, ready to be filled with life, is part of the Robot Operating System.

3.3 Robot Operating System

ROS is informally described as "an open source framework for getting robots to do things". Given that robots can essentially be seen as a distributed system of software and hardware components,

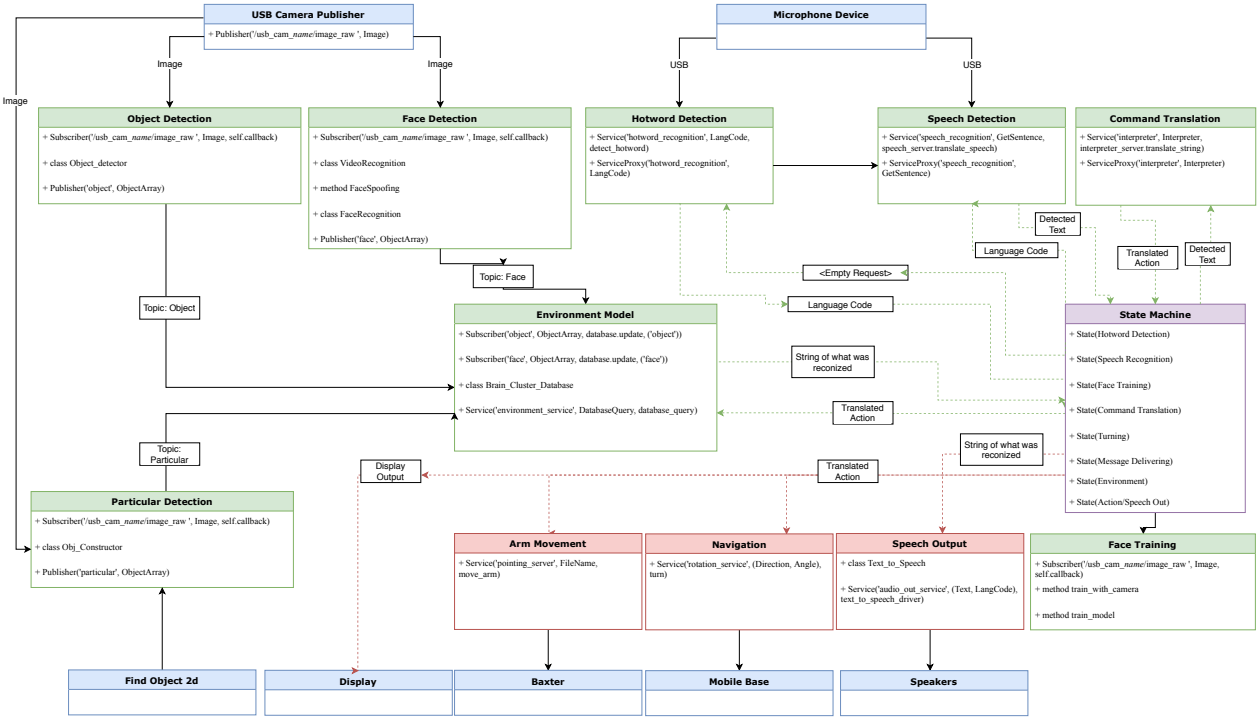


Figure 3.1: High Level System Design

each of which can be written to rely on firmware written in different programming languages, ROS serves a vital role in abstracting away much of the complexity in communicating across these different subsystems, enabling rapid prototyping of functionality and virtually eradicating barriers to creating a complex robotics project. Combined with the state machine, ROS was our other primary method of integrating and communicating across components to allow for persistent message passing through the system.

It is worth noting that, rather confusingly, the Robot Operating System is not in fact a genuine operating system. It does not have its own kernel and it does not supersede existing OS kernel privileges. Rather the ROS runtime "graph" is a peer-to-peer network of processes that are combined and integrated through an underlying communication infrastructure. ROS implements several different styles of communication, including synchronous RPC-style communication over services and asynchronous streaming of data over topics, both of which will be briefly described in this section.

ROS's modular and open source natural also makes it feasible to try out a wide variety of different packages. There are, of course, some difficulties in using and learning to profit from ROS's range of functionalities. ROS's online documentation is far from perfect and complete, when compared to a true OS or a common programming language. There can also be difficulties in working with hardware, frequent releases means old incompatible packages are scattered across the robotics community. However, even with a relatively steep learning curve, ROS played a vital role in allowing us to develop software and integrate and experiment with complex routines on DE NIRO, as it has in countless robotics projects across academia and industry.

As previously mentioned, we utilised two key message passing techniques that are implemented within ROS for both Synchronous and Asynchronous communications.

Asynchronous: Publisher-Subscriber Given the wide range of sensors and data streams that an advanced robot like DE NIRO is continuously collecting, it is necessary to have some route for software to request or send data to and from these sensors. This is precisely what the Publisher-Subscriber model enables us to do from within ROS. Any given node within our framework creates or selects a 'topic' which it writes to. Then, any other node is able to 'subscribe' to that topic through a callback function and waits for the relevant data to be transmitted. We use this substantially across each of our system components, for instance transmitting rotational information the the mobile base motor controller as well as subscribing to multiple camera feeds to run object and face detection routines.

Synchronous: Client-Service There is an important use case across our system for message passing across nodes, which allows the state machine to make informed decisions on where to 'go next' or what to do next. This can include action IDs after voice input has been interpreted, as well as rotational information required to turn to relevant objects. The client-service implementation within ROS enables us to seamlessly integrate each of our components into state machine and continuously change execution orders, providing us with a dynamic life-like flow within our demonstration. Often, the data transferred across nodes is simple parameters in the form of pre-defined, ROS consistent data types.

4 Development Approach

4.1 Scrum

When choosing between different development methodologies, we decided early on to follow the agile software development methodology with the scrum framework. Scrum per se concentrates more on project management methods, and not on software development alone, but it served very well as an overarching framework for our development process.

4.1.1 Meetings

We favoured regular team meetings, ensuring the entire team is always up-to-date given the current status of the project. Following this approach, we introduced three meeting times that were meant to ensure an adherence to continuous development cycles.

Daily check-in: The *daily check-in*, required every team member to post a brief 2-line description on a dedicated Slack channel (more in section 4.2) of every team member on weekdays, that would outline what they were planning to do on the project on that day. This did not mean we worked on the project every day. It was completely legitimate to state that there was no time to work on the project on a given day, but it meant a continuous flow of communication, which ensured a good distribution of information and helped us to stay in touch when we didn't see each other at busy times during the term.

All-hands meeting: We also had a weekly *all-hands meeting*, where every group member would share more comprehensive updates on the progress of his respective work-stream. These meetings also served as a platform to discuss potential roadblocks and presented a "safe space" for all of us to signal if we were stuck on a problem and needed help from another group member.

Sprint-update meeting: Every other week, the all-hands meeting was extended to a *sprint-update meeting*. In accordance with the agile methodology, we structured our work typically in 2-week sprints (described more in detail in section 4.1.2, to divide the work into manageable chunks. In sprint-update meetings, we would conclude a two-week sprint and set ambitious yet achievable goals for the next 2 weeks. When a sprint deadline was missed, we discussed if an adjustment of our overall timeline (see full GANTT Chart in the appendix, figure C.1).

Kick-off meeting: During this project we had three *kick-off meetings*, one at the beginning of each phase of the project, dictated by the project report deadlines. These meetings were typically between 2-3 hours long and allowed for a more large scale discussion, setting overall goals and deadlines for the project phase, assigning work-streams, developing rough program specifications, outlining dependencies between works-streams and exploring if our overall system architecture would allow for the additions and changes to come. These meetings were particularly helpful for us to get everybody on the same page concerning the overall goal we needed to work towards, effectively unifying the team behind a common "vision".

4.1.2 Development Cycles

As eluded to above, we typically divided our development work into manageable chunks of 2 weeks. The "Cost of Change" graph in figure 4.1 shows the cost of change curve by Barry Boehm (1981), described in "Software Engineering Economics"[4]. The graph shows that while making changes

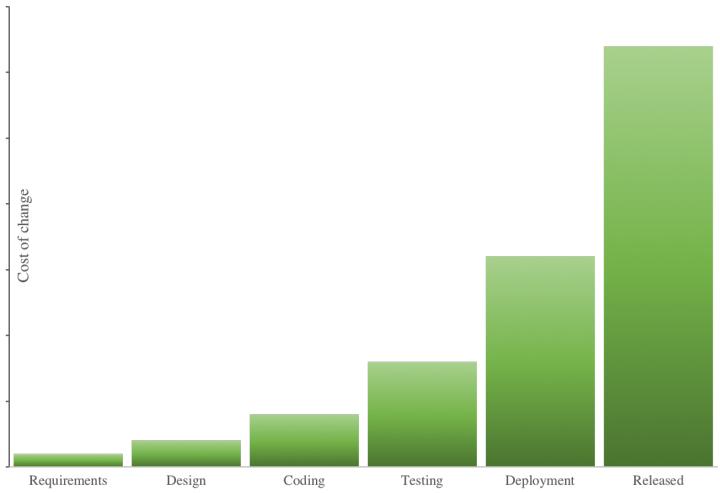


Figure 4.1: Cost of change in software development, by development cycle

early in the development process is relatively "cheap" or easy, introducing changes later on becomes more costly in terms of time and effort. Still it is nearly impossible to clearly define all software features right from the start. Agile therefore takes this cost of change view, but condenses it to many small iterative cycles, each of which goes through the phases indicated in the graph. The list in Appendix A briefly summarizes our sprints over the last few months.

4.1.3 Pair-programming

Particularly in the beginning of the project we split work into pairs. This way we tried to ensure that if one of us got stuck on a specific problem, the corresponding sub-team partner would be able to help out. We also hoped that this approach would help us make fewer mistakes in the development processes. As the project matured we slowly changed this structure, however, since each of us started feeling more comfortable writing code on their own, and we were able to tackle a larger number of challenges at once.

4.2 Project Management Tools

Choosing the right project management tools was vital in helping us move forward with the development process.

4.2.1 Communication Tools

Slack We selected Slack as our primary communications tool. We created dedicated *channels* on Slack for e.g. "minutes-and-admin", "links-and-resources", and "project-reports". This helped divide communication into logical streams and greatly reduced the clutter that built up in a single chat conversation. The "check-in" channel was dedicated for our daily check-in messages as mentioned in section 4.1.1. See figure 4.2 for an example.

Email, calendars, other "standard" tools Email, calendar invites and other "standard" tools of communication mostly served us in communicating externally, to set up meetings with potential project supervisors, booking rooms for testing DE NIRO in an external environment etc. Internally such tools were made completely redundant by Slack. We also maintained a WhatsApp group with

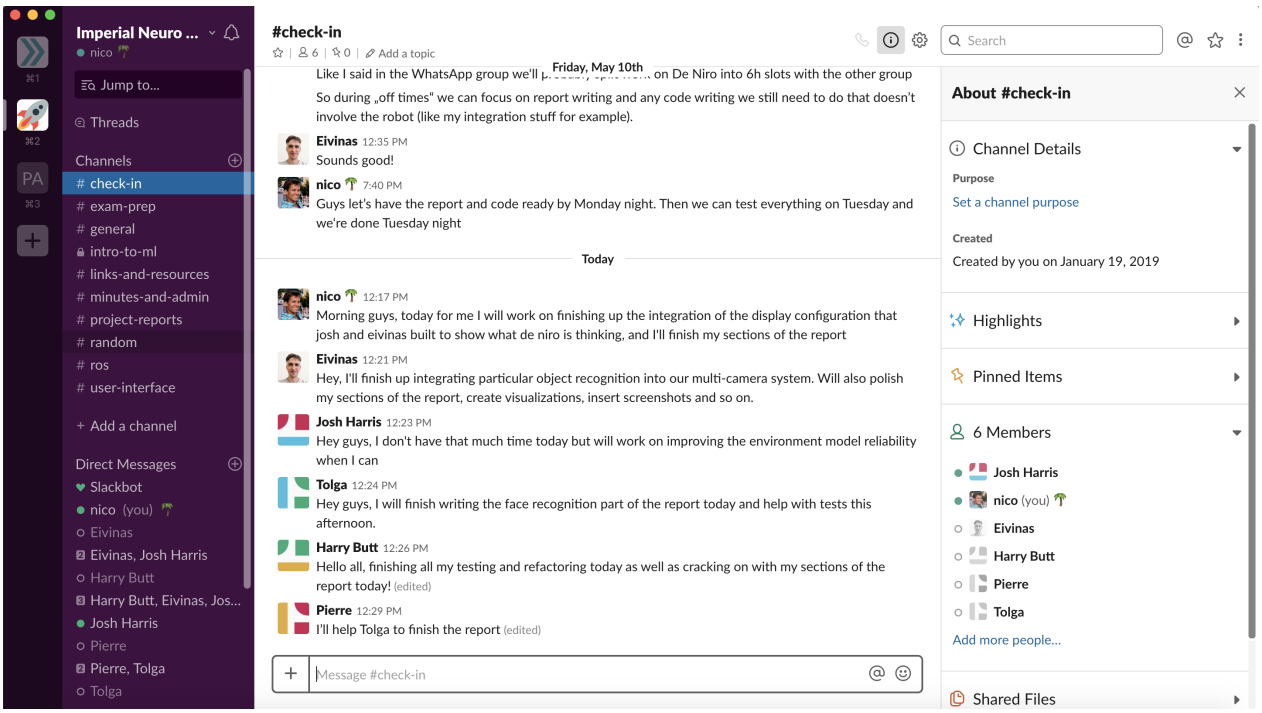


Figure 4.2: Screenshot of our Slack environment (Check-In Channel)

our supervisor Petar and the other group working on DE NIRO, to schedule ad-hoc meetings, and coordinate the use of the robot between the groups.

4.2.2 Managing backlog

We quickly realized that a software development project as complex as programming a robot to play "hide-and-seek" called for a tool to effectively manage the backlog of new features to be developed, bugs to be fixed, and integration work to be done. We landed on **Trello**, a project management tool that favours the Kan-Ban approach, which helps development teams to focus only on the immediate development work that lies ahead of them, instead of being overburdened with an insurmountable number of tasks.

We found Trello had good traction within the group during the early phases of our project when there were many small tasks we were often conducting independently. However, as we moved onto the main integration tasks with us all around the robot we found Trello was of less use and generally slipped out of our routine.

4.3 Git

Our version control system of choice was the GitLab infrastructure provided by Imperial College. None of us had extensively worked with Git before and already the simple task of submitting a commit after the completion of one logical unit of code proved to be a difficult habit to build.

We truly discovered the usefulness of Git the hard way, when Dr. Kormushev asked us to present our intermediary progress to a group of visitors from Singapore the following week. At that point in time we had a stable version of our code running on the Robot Brain Cluster. We committed

and pushed that code to Git, and kept working on the project until the visitors arrived. As we had worked on additional features in the meantime, we had introduced bugs into the code that we only encountered minutes before the scheduled brief presentation. Since none of us had spent enough time familiarizing ourselves with Git, we were unable to **git checkout** the stable commit, causing the robot to crash mid-demonstration.

Having learned from this rather unpleasant and somewhat embarrassing experience, we henceforth ensured we would always be able to checkout a stable and well-tested commit on our system should a similar situation arise.

4.4 Testing

Our learnings from Git were tightly linked to our learnings involving testing. The combination of distributed applications and specific hardware involved in a robotics project meant we saw integration and system-testing as particularly relevant, and required us to be extra diligent with testing. We decided to split testing into two major components:

First, we unit tested each node that was independent from robot hardware individually but with greater focus on testing the input we need the node to be able to accept from other nodes and the output the node needs to provide. *Second*, we implemented minimum viable nodes on the hardware we were going to use as early as possible. This allowed us to iteratively do integration tests as each new node was added. The culmination of successful integration tests would pass a scenario based system test for a final demo of our project (essentially our own user story).

4.4.1 Unit Testing and Test-Driven Development

The comparatively easy part of testing was to develop unit tests for each component, adhering by the principles of **test-driven development (TDD)** (see Appendix D for unit test coverage). In TDD, we start off by writing a failing test and consequently enter a so-called "red state". To get to a "green state", we consequently write a minimum version of our function/method/component that passes the test. We then add the desired functionality to our method and constantly verify whether we still pass our tests.

We started off by writing tests as an after-thought, something that was only required by the course-specifications, but quickly realized their value. The more complex our code base grew to be, the more invaluable our unit-tests became. Whenever we decided to refactor a complicated piece of code, we were able to ensure correct functionality of individual components by merely running our tests.

In retrospect, one change that could have helped us design even more effective tests, would have been to adhere to a **"strategy pattern"** design approach, decoupling objects as much as possible, which is particularly suitable when coupled with mock-objects, allowing for very specific tests on singular objects. This approach would have allowed us to automate certain system tests which we ended up carrying out manually instead. As we only learned about the strategy pattern at a later stage in the development process, we decided to stick to our existing approach, more closely related to the **"template method"** to ensure we hit our deadlines.

Unit Testing with ROS: One brief mention should also be made on the Unit-Testing with ROS. As many logical units in our code involve publishers, subscribers, services, clients, actions,

etc., which are native to ROS, unit-testing our code was somewhat more complicated than unit-testing a regular software application. After some research and a lot of trial and error, we figured out that each bit of code related to ROS required to set up a simulated ROS environment with *rospy* that would set up a ROS master and a ROS node in which the code could run. An example of that structure can be seen in Appendix D.

4.4.2 System Testing

After ensuring that our unit tests passed to our satisfaction, we would move on to system testing.

To identify bugs early on, we developed standardized routines as system tests that we expected the robot to be able to perform after every major update, testing as many different invocations of our code as possible. Initially these routines were fairly simple (e.g. "what can you see" / "who can you see", merely triggering a voice-out for the robot's camera feed).

The "What can you see?"-Routine also doubled as our first Product MVP. Early on we had agreed that we needed to hit this intermediary goal, because it ensured that we understood ROS, and Robot DE NIRO well enough, in order to start building a more complex architecture.

Over time, our system tests grew more complex, involving the robot turning, pointing, identifying ownership and learning new faces on the fly. We achieved some degree of automation in performing these tests, by providing the robot with certain standardized queries that bypassed the voice-input (after ensuring that the voice-input worked flawlessly), but most procedures involved human interaction to make sure not only the software, but also the hardware worked correctly.

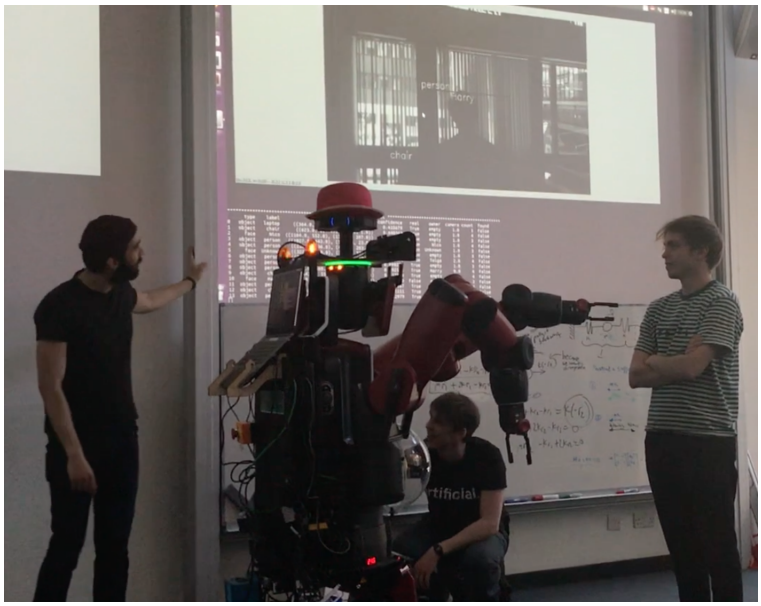


Figure 4.3: Environment Model and State Machine System Testing

4.4.3 Code Coverage

For individual nodes we continued to reach coverage rates like the 87% statement coverage and 86% branch coverage we saw after implementing our MVP. However, as we moved towards building the full system on Robot DE NIRO hardware we increasingly found system testing a more appropriate method for the integration nodes. This was for three reasons. First, hardware and concurrency issues made up the vast majority of our errors and would only reveal themselves when we tested the nodes interacting with one another on the robot. Second, the number of possible paths through a routine grew exponentially given all the possible combinations of input/output/objects etc. meaning normal unittest code coverage metrics showed coverage even when we knew there were many edge cases we likely still needed to explore. Finally, truly testing new functionality (such as multi-cameras in the environment model) increasingly became impossible without running on the actual hardware.

4.5 Key Learnings

Aside from the many points around, tools, unit-testing, software architecture etc. there are a few key learnings we are taking away from the choices we made in the development process, which irrespective of the ultimate design choices and development methodologies, we believe are vital to the success of a software development project:

Time-management: Manage time well. Adding or changing new features near a deadline becomes increasingly stressful, and error-prone. Quick-fixes begin to dominate over "future-safe" solutions. A **technological debt** is built up that the team has to deal with at a later stage, making it harder to add new pieces of code each time.

Manage backlog: Clearly structure upcoming to-dos so that (1) nothing falls off the wagon and (2), responsibilities are easily assignable.

Test-driven development: Commit in a **green state**. Be ready to deploy at any time. Don't overwrite a stable code base with a new one that has only one small additional feature just before a demo.

5 Product Features

5.1 Brain Cluster

Owners: Entire Project Team

Introduction The starting point for our project was the idea that by creating a framework for easily adding remote computers (a "Brain Cluster") into DE NIRO's ROS environment we could significantly expand its capabilities into computationally intensive tasks such as image analysis.

The technical aspect of the "Brain Cluster" was mostly us getting up to speed with the capabilities and relevant packages in ROS than writing code. Therefore, most our time was spent on:

- Designing an extensible framework
- Implementation of the framework

The Framework Starting out we had little idea the final number of CPUs and GPUs our project would require (up to a maximum of c.20 desktops we had available). This meant the "Brain Cluster" framework had to be easily extensible, allowing us to add a new computers and nodes to the network as our scope increased. We settled on two design principles:

1. **Remote launch** meant that we could use ROS launch files from the laptop on DE NIRO to launch all the nodes on the "Brain Cluster" computers, removing the need for any contact or UI on the computers themselves. This stopped us from creating specific file structures, environments etc. on individual computers. Remote launch was actually a novel approach, that had not been tried by previous groups working on DE NIRO before.

2. **Remote installation** meant that to bring a new computer into the "Brain Cluster" you could simply plug the computer into the network, remotely copy a pre-existing package onto that computer, compile the package and it would be ready. This forced us to put everything we needed for a node (code, pre-trained weights, bash scripts for remote launch etc.) into one ROS package which was not tailored to any one desktop.

```
<launch>
    <machine name="robin_6" address="192.168.0.228" password="XXXXX"
            env-loader="~/face_recog/cluster.sh" user="robin" timeout="30" > </
    machine>
    <node machine="robin_6" name="face_dect_left" pkg="face_spoofing" type="
    recognition_runner_left.py" cwd="node" output='screen' args="usb_cam_left/
    image_raw 0" respawn="true"/>
    <node machine="robin_6" name="face_dect_middle" pkg="face_spoofing" type="
    recognition_runner_left.py" cwd="node" output='screen' args="usb_cam_middle/
    image_raw 1" respawn="true"/>
    <node machine="robin_6" name="face_dect_right" pkg="face_spoofing" type="
    recognition_runner_left.py" cwd="node" output='screen' args="usb_cam_right/
    image_raw 2" respawn="true"/>
</launch>
```

Figure 5.1: Example Remote Launch File [Face Detection]

Implementation Practically setting up the "Brain Cluster" was primarily a steep learning curve to understand the specifics of the DE NIRO ROS environment, networking and how to best run ROS over a network of computers. After a brief investigation of options such as running in AWS or creating a wireless network, we opted for simplicity and bandwidth by using a wired LAN. The other main challenge was creating bash scripts and launch files that could automate the process of connecting to the ROS MASTER on DE NIRO and creating nodes.

Key Learnings Whilst development was largely successful, there are a few corners cut which could have improved the overall architecture. Firstly, a wired network has significant drawbacks and shifting to a wireless network (as discussed further in Appendix B), whilst improving the compression of image data to reduce required bandwidth would be key to combining the "Brain Cluster" with greater movement by DE NIRO. Second, investigating the potential of using a "Two MASTER" approach where the "Brain Cluster" interacts internally with a separate ROS MASTER from DE NIRO. This approach would have the benefit of keeping much of the communication between "Brain Cluster" nodes off the main ROS MASTER.

5.2 Object Recognition

Owners: Joshua Harris and Eivinas Buktus

Introduction An underlying aim of our project (a "Brain Cluster") and our Hide and Seek demonstration was giving DE NIRO the ability to categorise its environment and use this information to inform its actions. We see object recognition as a first step in moving from a purely physical understanding of the world ("there is something blocking my path") to a semantic one ("there is a person blocking my path and he is Nico who I have a message for" or "that is an imperial college ID so they are authorised to operate me"). In this way object recognition enables us to code instructions that utilise and build on a far richer picture of the environment DE NIRO finds itself in. For our project, implementing object recognition involved three components:

- Generic Object Recognition - e.g. book
- Particular Object Recognition - e.g. Lord of the Rings
- Combining Objects into a Coherent View of the Environment - e.g. Tolga has the Lord of the Rings

Generic Object Recognition We investigated a number of potential solutions and forms of object recognition, including: YOLO (You Only Look Once), RetinaNet, TensorFlow Object Detection API, pytessarct (Optical Character Recognition) and Google Cloud Vision. We chose to implement two prototypes, one of YOLOv2 and one of RetinaNet, both of which were successful with similar performance on test videos. On the back of this we decided our final implementation of generic object recognition would use the YOLO framework [8] pre-trained on the COCO dataset of 80 common objects [10]. We selected YOLO over RetinaNet due to it being lighter weight (crucial as we could be running up to 6 instances, one for each camera) and there being an existing open source python implementation with pre-trained weights.

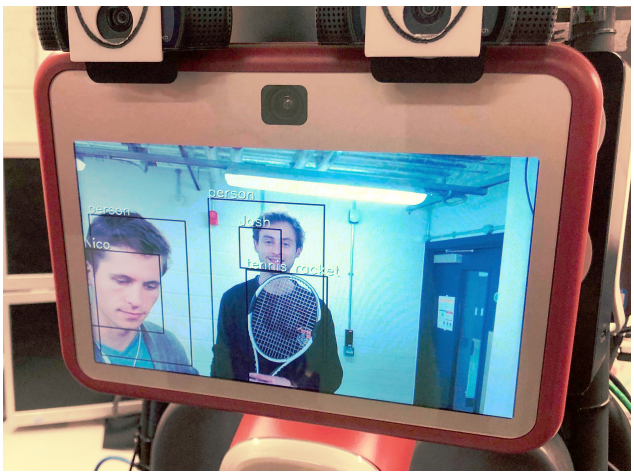


Figure 5.2: Screenshot of Generic Object Detection - Tennis Racket

Implementation Installing the YOLOv2 architecture itself was relatively straight forward using the *darkflow* python implementation and TensorFlow. Through an iterative process we developed a ROS node that can be remotely launched by passing a specific camera topic and ID and calls YOLO recognition. The results from the YOLO recognition are parsed into a custom ROS message we created and published with the ID of the camera. This setup allowed us to have one self contained ROS package that could be implemented for any camera on any computer in the "Brain Cluster".

We also found issues with transmitting image data from all 6 Logitech C920 cameras. We reduced our input to only 3 cameras but still saw intermittent overloading. Therefore, as image quality is clearly key for successful object recognition; rather than degrading the quality we instead lowered the frame rate of the cameras from c.30 frames a second to below 10. We saw this as an acceptable trade off as any one instance of our object detection node could only process a frame every c.0.5-1 seconds. Another issue was the computational intensity of the nodes, leading us to split up object recognition onto two dedicated computers in the "Brain Cluster". For us overcoming these issues was another illustration of the benefit of developing self-contained ROS packages and a flexible brain cluster architecture.

Issues and Improvements Due to the greater availability of pre-trained weights and python packages we continued to use the YOLOv2 as our object recognition package rather than the most recent YOLOv3 architecture. Although the performance of YOLOv2 architecture is adequate for our tasks, there remains some inconsistency, particularly when objects are at a distance. Therefore, if we had more time we would certainly investigate whether it is worth training the v3 architecture ourselves to improve performance.

Second, the range of 80 objects in the COCO dataset is only partially appropriate for a robot like DE NIRO. Aeroplanes and Zebras were not frequently sighted in our lab and would be unlikely to make the list of the 80 most relevant objects if we were to choose ourselves. Removing irrelevant objects and re-training the model with a tailored set of objects would allow us to improve our understanding of the environment whilst keeping the computational complexity of the ROS node the same.

Particular Object Recognition As an extension of object recognition we also decided to implement a system that would be able to recognize particular objects. These objects include specific book covers (e.g. "Lord of the Rings"), logos (e.g. "Logitech") and other surfaces that have clearly

defined edges (e.g. "Imperial College ID"). This capability enhances DE NIRO's understanding of the surrounding world since there is an almost unlimited amount of new things that DE NIRO can learn/be taught to recognize (unlike using YOLO with the somewhat limited 80 object COCO dataset mentioned above).

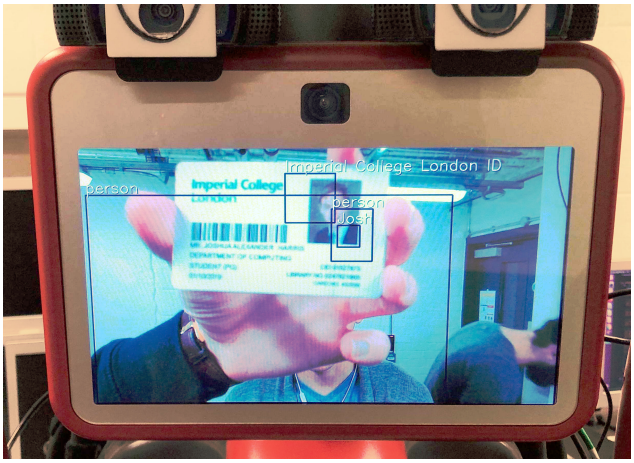


Figure 5.3: Screenshot of Particular Object Detection - Imperial College ID

To implement this we used a ROS package called *find_object_2d*, which lets one use feature detection algorithms like SIFT, SURF, FAST, BRIEF [17]. Essentially, these algorithms take an image and produce a kind of 2D point map representing interesting features of the image like edges and corners. These features are then matched with features of objects in the database. In terms of the specific feature detector, we decided to use SURF as it is said to be both relatively fast and accurate [18].

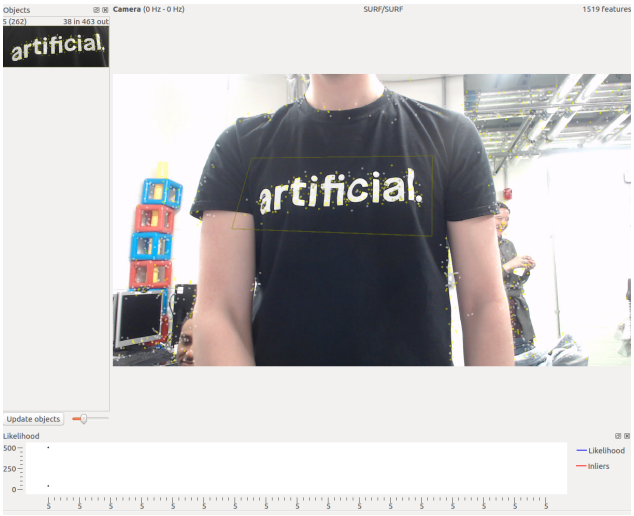


Figure 5.4: find_object_2d GUI

One difficulty that we encountered was to make *find_object_2d* work with our multi-camera setup. We solved this by launching multiple *find_object_2d* nodes for each camera and remapping

the topic that they publish to (i.e. *objects*) to three different topics. Then our own written ROS nodes subscribe to these topics and publish the recognized particular objects in the format that the *Environment Model* (see 5.4) understands.

5.3 Face Recognition

Owners: Tolga Dur and Jiacheng Wang

Introduction The Face Recognition package is a crucial part of the Hide-And-Seek demonstration and providing DE NIRO with a contextual understanding of the local environment. Here, apart from the simple face recognition that is run on three cameras through the demonstration routine to assign objects to specific people (owners), the spoofing detection and live training proved to be very useful in showcasing DE NIRO's extended processing capabilities.

Implementation As can be seen from Figure 5.5 , our package consists of the "ModelTrainer", and "VideoRecognition" classes. The prior is responsible for training the model, while the latter for the actual detection, recognition, and spoofing of faces it gets from subscribing to a relevant camera node. The result is then published to the environment model from which queries can be made.

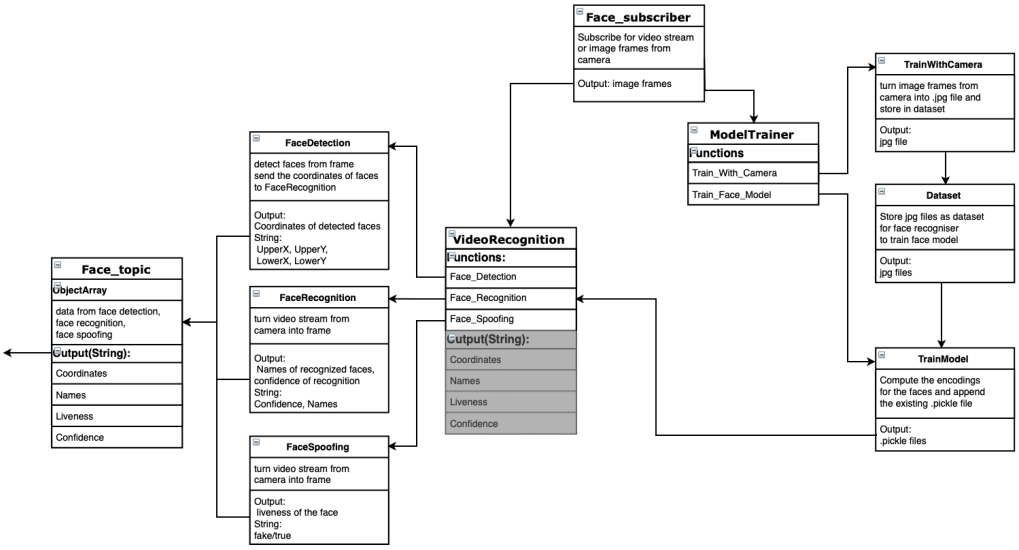


Figure 5.5: UML diagram of the face- and spoofing recognition package.

The Face Recognition works with video input from the same three cameras as Object Recognition. The video stream frames are resized and sent to the relevant API of dlib (the framework that supports DE NIRO's existing face recognition abilities) to get the coordinates of all faces in the frame. Subsequently, the recognized faces are encoded with Haar Cascade[1]. We decided against using a Convolutional Neural Network based approach despite its higher accuracy as training is an extensive process not suitable for a live demonstration. Subsequently, we utilise a K-nearest

[1]Algorithm used for object detection based on features proposed by Paul Viola and Micheal Jones (2001).

neighbours approach. If any face in the trained model matches with an accuracy of more than 50% percent, the name of the face is published to the environment model together with the coordinates and Boolean specifying, to our best estimate, whether it is an actual person or a static picture. If on the other hand the encoding has a confidence of less than 50% with every picture in the trained model, 'UNKNOWN' is published.

Figure 5.6: Screenshot of Face Detection - Josh and Eivinas

Difficulties Initially, we had decided against using the dlib library due to slow reaction times, limited recognition of unknown faces and the limited view resulting from the computationally expensive implementation of the package on the Kinect camera. Instead, we implemented a prototype version of face recognition with the native approach of the OpenCV library, with confidence interval computations used for the recognition of untrained faces. However, this approach relied very heavily on expensive image prepossessing such as face alignments. Even then, results were unstable and struggled to recognize faces we had trained on hundreds of pictures. Hence, we decided instead to fix the key issues we had with the dlib library. First, we found a way to implement the dlib library with the same sensors as the Object Recognition Package, assisting with limited view problems and smoothing the integration process. Further, we figured out a way to compute confidence intervals by counting the amount of matches of the frame with each frame in the trained model and then dividing the count by the number of pictures in the database for each face which allowed us to recognize unknown faces precisely.

Extended Capabilities We initially considered three possible extensions to Face Detection:

- Spoofing detection on faces was implemented successfully, however for objects preliminary research deemed it unreliable and of little value to our final demonstration
- Pose Estimation, and
- Emotion Detection - both of which were deemed to be of little benefit to our demo routine and future work on DE NIRO

Therefore, we decided to instead focus on implementing a live training capability which was not planed initially and was accomplished successfully.

Spoofing Detection We attempted several different strategies to achieve a somewhat robust and useful spoofing detection for DE NIRO. Initially, we were heavily relying on a open source library utilising deep learning to detect the difference between videos shown to the camera through a phone screen and live footage. However, since the library was trained in very specific lighting conditions it ultimately proved to be an unreliable face spoofing detector.

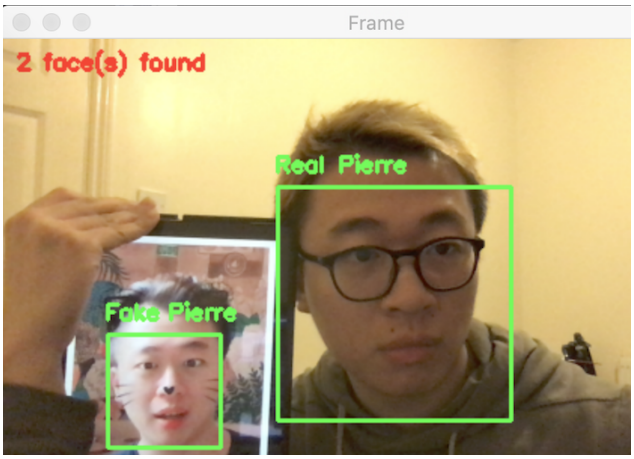


Figure 5.7: Screenshot of Face Recognition and Spoofing - Pierre

We finally decided on a more pragmatic solution of analysing subtle movements of faces to distinguish between static pictures and videos/live-streams. We decided to focus on eye-movement which practically meant searching for any eye movement within a video frame in a given period of time. This turned out to work much more stably in real-time conditions and classified faces more accurately than all deep learning based approaches we have tested.

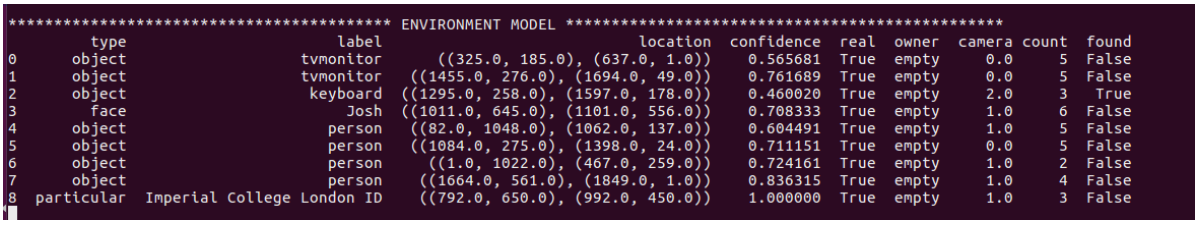
Our solution does, however, come with some shortcomings. First, we have to wait several seconds for possible eye-movements before classifying a face as fake. Furthermore, one difficulty was the selection of appropriate hyper parameters. One illustration of this difficulty can be illustrated by the example of the EAR parameter. Initially, our solution classified Asian people as always having their eyes closed. After tuning the EAR parameter to incorporate this, however, it thought that Caucasian peoples eyes were always open which lead us to do many tests before finding the perfect EAR parameter.

Live Training of New Faces Unfortunately, we could not find a viable solution for live training of faces with OpenCV or dlib. The only solutions that we found for live training of faces were not using either of the two libraries and were based on very complicated software and hardware. Fortunately, just saving 25-50 frames from the video stream as pictures in the database and then training the model on static pictures as normally done, worked surprisingly well. However, we then had the problem of having to retrain the entire model whenever a face was trained. The solution to this was found after carefully analysing how the pickle file generated by dlib looked like, and consisted of artificially deleting/adding face encodings to the file regarding the specific face want to train.

5.4 Environment Model

Owners: Joshua Harris and Eivinas Buktus

One of the main components of our project is the *Environment Model* (initially called "Brain Cluster Database"). This is where most of the "thinking" happens and this is what provides DE NIRO with situational awareness that is usually lacking in simple object or facial recognition systems.



```
************************************** ENVIRONMENT MODEL ***********************************************
        type                      label                                location  confidence  real  owner  camera count  found
0     object                  tvmonitor      ((325.0, 185.0), (637.0, 1.0))    0.565681  True  empty     0.0     5  False
1     object                  tvmonitor   ((1455.0, 276.0), (1694.0, 49.0))    0.761689  True  empty     0.0     5  False
2     object                   keyboard  ((1295.0, 258.0), (1597.0, 178.0))    0.460020  True  empty     2.0     3   True
3       face                       Josh  ((1011.0, 645.0), (1101.0, 556.0))    0.708333  True  empty     1.0     6  False
4     object                     person   ((82.0, 1048.0), (1062.0, 137.0))    0.604491  True  empty     1.0     5  False
5     object                     person   ((1084.0, 275.0), (1398.0, 24.0))    0.711151  True  empty     0.0     5  False
6     object                     person     ((1.0, 1022.0), (467.0, 259.0))    0.724161  True  empty     1.0     2  False
7     object                     person    ((1664.0, 561.0), (1849.0, 1.0))    0.836315  True  empty     1.0     4  False
8  particular  Imperial College London ID    ((792.0, 650.0), (992.0, 450.0))    1.000000  True  empty     1.0     3  False
```

Figure 5.8: Environment Model screenshot

The *Environment Model* aggregates information that is coming from three subsystems (see 5.2 and 5.3):

1. Object recognition (e.g. "person", "backpack")
2. Particular object recognition (e.g. "Imperial College ID", "The Lord of The Rings")
3. Face recognition (e.g. "Tolga", "Josh")

The information from these subsystems includes the labels, locations, confidence levels of the things being recognized. The *Environment Model* aggregates the information into a meaningful whole taking into account spatial and temporal contiguity (i.e. how close things are in terms of space and time), ownership relations (i.e. who is holding what) and the fact that we are using multiple cameras. The model of the world that is created can then be queried by any other ROS node.

Spatial and Temporal Contiguity The first thing that *Environment Model* does in creating a model of the surroundings is that it takes into account spatial and temporal contiguity of objects. This makes our world model more robust. In terms of spatial contiguity, the idea is that if DE NIRO has seen the same kind of object in a similar location before, it can deduce that a particular object has moved instead of thinking that a new object replaced the old one or that there are now two objects. For instance, if recently there has been a football recognized at some location and then it moves a few pixels to the left in the next frame, we would simply update the location of the previously seen football.

Similarly, temporal contiguity is taken into account by counting which objects and faces have appeared in recent frames. For instance, if a face appears in a frame, its counter gets incremented, otherwise it gets decremented. This face becomes part of the model of the environment only if its counter reaches a certain threshold. This deals well with the fact that object and face recognition systems are not robust enough to reliably detect the same things in consecutive frames. It also deals

with the fact that sometimes these systems detect things that are simply not there (e.g. we would get a traffic light or an oven in the lab sometimes). By requiring the same things to be detected over multiple frames, such "quirks" are rightfully ignored and do not become what our system thinks is the ground truth of world.

Ownership Allocation The second big thing that *Environment Model* does is that it determines who is holding what, i.e. allocates ownership. First we map bodies (as provided by object recognition) to faces (provided by face recognition). This is done by seeing which bounding boxes belonging to bodies overlap with face bounding boxes. Now knowing that the bounding box of the body belongs to some person (e.g. Josh) and we can see whether this bounding box overlaps with objects to some degree. If they do, that means that in the environment model the person is the owner of that object.

Multiple Cameras One further important capability that our system supports is multiple cameras. The information coming in from all the different cameras about objects and faces is combined in the *Environment Model* in a meaningful way. The cameras are simply treated like another dimension in space. So for instance, even if we have have two footballs being detected with similar pixel coordinates, there will be two of them in our database if these footballs are detected from different cameras.

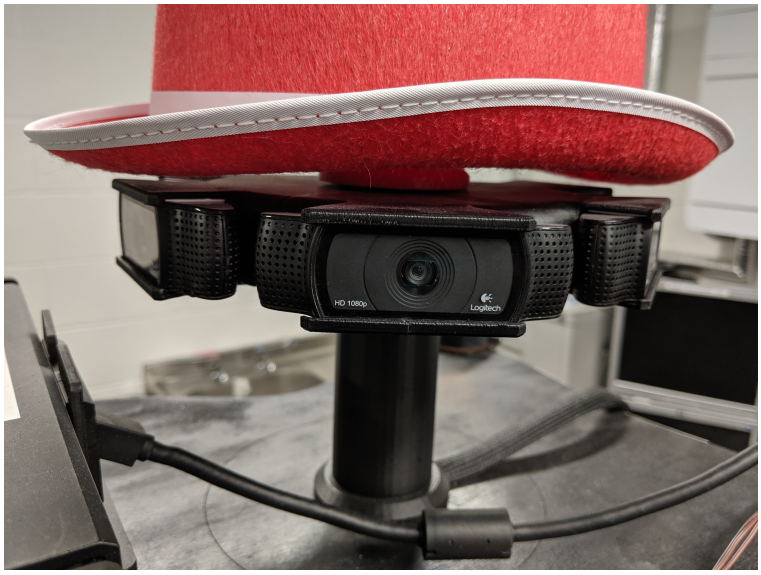


Figure 5.9: DE NIRO's Multi-Camera Rig

Since DE NIRO has a 360 degree camera rig, gathering information from these cameras provides a much richer understanding about how DE NIRO is situated in his environment than using one front camera. For instance, DE NIRO is able to know and say things like "Nico is behind me on my left".

Integration The *Environment Model* provides the *environment_service* by which it can be queried. At the moment it simply returns all of the data that is thought to be the "ground truth" of the

current state of the world around DE NIRO.
Going further, this could be extended to more specific queries (e.g. return all the faces in a specific camera or return a list of objects that, say, Josh has). At the moment in our software framework such inferences are made by the state machine (see 5.5). It receives the world model from *environment_service* that includes information about all the cameras, all the objects and faces, all ownership relations, locations and so on.

5.5 State machine

Owner: Nicolas Tobis

If the Environment Model is the brain of the whole operation, the state machine is the motor that keeps things going. A state machine allows the programmer to separate a robot's tasks into logical states that are executed sequentially. ROS provides the SMACH package, a widely-applied, ready-to-use state-machine, which holds the state machine container class as well as state classes that are added to the state machine container, once the robot is launched. Each state has a specific responsibility. E.g. the listening state will launch the speech recognition, and the Interpreter state will translate a sentence into an actionable command.

States can also pass information amongst each other, and irrelevant information is not made available to a state. This makes the SMACH package particularly handy, as it allows for very precise passing of information, as opposed to making global variables available to all states.

5.5.1 Design Choices

State machines are often used for comparatively linear paths of execution. However, early on we decided to make our system as user-friendly as possible, meaning feedback-loops if user-input was incorrect, and branches depending on the decision a user made. Thus, all decisions we made in the design process of developing the state-machine focused on usability first, with the end-goal of developing a fail-safe, closed system that the user could seamlessly interact with. As a consequence, we needed to make the state machine more flexible, and allow it to branch off, if need be. Figure 5.10 shows our final state machine's flow of control.

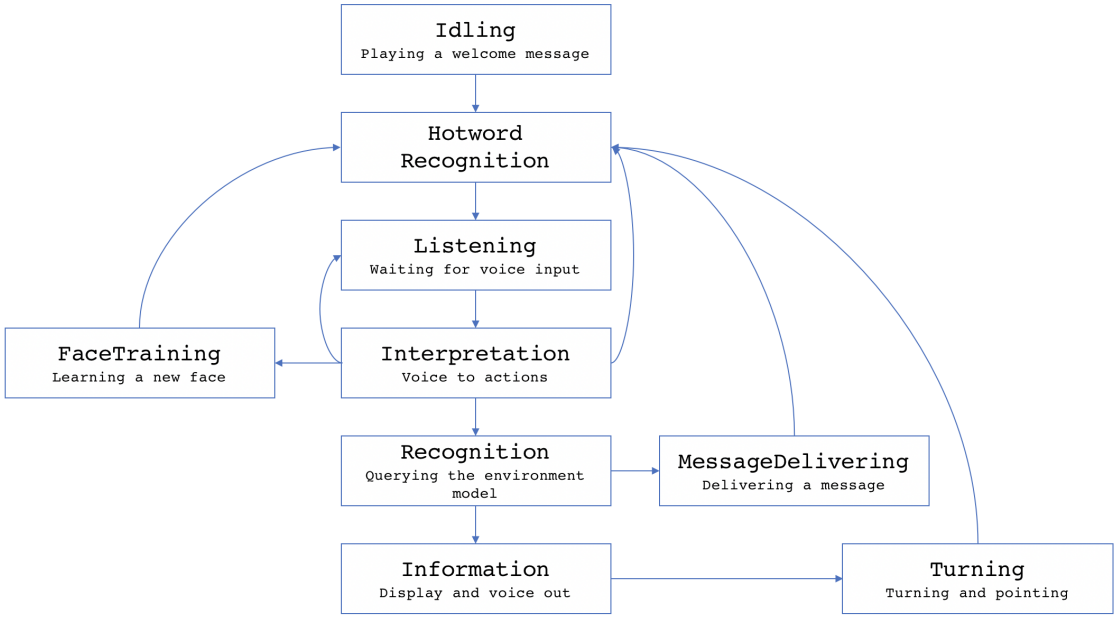


Figure 5.10: State machine UML Diagram

While states are internally linked and able to pass information to each other, they communicate with other ROS-nodes via services. A service is called on one of the other running ROS-nodes, e.g. Speech Recognition, which provides the state with a piece of information, e.g. a string that was converted from human speech.

5.5.2 States

Idling: The Idling state is just a simple start-up state, prompting the robot to play a welcome message, when it is booted.

Hotword Recognition: The Hotword waits until the Hotword "Hey DE NIRO" is said. It is the starting point for all interactions with DE NIRO. Similarly to "Alexa" or "Hey Siri", this locks the Audio Input from any other input around the robot preventing it to perform random actions as a result of involuntary speech commands. While DE NIRO is waiting for a Hotword it will also play a "waiting for input" animation on the screen.

Speech Recognition: The Speech Recognition state always follows the Hotword Recognition state. It waits for the user to give a speech command, which once its given is shown to the user as a string on the screen as a feedback mechanism.

Interpretation: The Interpretation state takes the string, and converts it into an action. Most of the states that follow the interpretation state depend on it's output. The interpretation state will decide, if the robot needs to e.g. recognize a specific person, take an inventory of everything it sees in the room, deliver a message to a person, or learn someone new on the fly. Consequently, the Interpretation state can call one of 4 states: Hotword Recognition, Listening, Recognition, and Face Training.

Recognition: The Recognition state queries the robot brain cluster, and more specifically the environment model. It receives a snapshot from the environment model about the current state of the world and branches into either the Information state or the Message Delivering state, dependent on the information it received from the Interpretation state. The information it received from the environment model is consequently made available to those two states.

Information: The Information state does the heavy lifting of transforming what the robot believes the state of the world is into outputs that humans can understand. It queries the text-to-speech conversion to construct a coherent string based on the query it received from the interpretation state, e.g. "Where is the ball?", together with the information from the recognition state, i.e. the current state of the world. It also sends the relevant parameters to the screen of the robot, so that the user knows why the robot classified certain objects and people as it did. After completion, the Information State will call the Turning state.

Turning: The Turning State is always called after the Information state. However, it is only executed if it makes sense for the robot to turn, i.e. once the robot is asked to look for an object or a person that it locates on one of its back cameras. The Turning State is also responsible for pointing, i.e. pointing towards a person or an object, if asked where the item in question is located. Thus, in retrospect, "Movement" might have been a more fitting name for this state. The Turning state always returns to the Hotword Recognition state.

FaceTraining: The Face Training state will prompt a user for his or her name, and learn the user's face. Much like a human, it will then remember that person's name, and correctly identify the person in subsequent queries. Once successful, the Face Training state will return to the Hotword Recognition state.

Message Delivering: The message delivering state is designed to showcase the most of DE NIRO's capabilities in one. It requires that DE NIRO has taken a message before (in the Interpretation State) for a specific person. Once called, from the Recognition state, the Message Delivery state has access to the current state of the world provided by the Recognition state, which queried the Environment Model for a current state of the world. It then performs three checks:

1. Can it see that person that DE NIRO previously took a message for (Face Detection)
2. Is that person real (spoofing detection)
3. Is that person able to present a valid Imperial College ID (Special Object Recognition using a point cloud)

If all three checks are passed, the Message Delivering state will playback the message and return to the Hotword Recognition state.

5.6 User Interface

Owners: Nicolas Tobis and Harry Butt

The User Interface forms the backbone of our game of hide and seek. Once the state machine has been launched, every command that follows to change the flow of control is dependent on user interactions, making both ease of use and reliability key factors for every design decision made (as

they are for any effective user interface).

Initially, there was a decision to be made regarding whether we wanted to influence the flow of control through a graphical user interface, connected to DE NIRO through a smartphone or tablet, or a completely voice-led interactions. Given the work that previous groups had done on voice interactions, limitations that arise from platform specific implementations with a GUI, and most importantly the core purpose of DE NIRO to understand and research human interactions we decided that an interface that recognised and processed voice input and used DE NIRO's on board output capabilities (screen, microphone) would be the most effective interface design.

5.6.1 Hotword detection

Introduction: Accurate hotword detection was one of our goals early on. The natural way for users to interact with many of their devices today is to call out a word that is easily remembered ("Hey Siri", "Alexa", "Hey Google"), but distinct enough from everyday words that could be spoken around the device, activating it by accident.

Implementation: In essence, we explored three paths of implementing a Hotword for DE NIRO:

1. Use a typical speech recognition package and perform string comparison on the output
2. Reprogram an Amazon Alexa
3. Research if existing hotword recognition packages exist

Using speech recognition to also perform hotword recognition proved to be slow and unreliable. We discarded it as a possibility early on. After investigating the programmability of the Amazon Alexa we found that it is a closed system to a large extent, not open to a lot of modification. It allows for three different hotwords ("Alexa", "Echo", and "Computer"), none of which seemed particularly appropriate to use on DE NIRO.

Seemingly the only open-source project of its kind, the Snowboy package allowed us to train a neural network on the user's voice saying a specific hotword. It is extremely robust even after training the model on only three iterations of saying the hotword. It works offline, and allows for training the model in 20 different languages. Finally, it is very computationally efficient, using signal interrupts to signal a hotword was recognized, rather than expensive busy waiting. Implementation was somewhat complicated, by the fact that the typical **pip install** did not work, which meant setting up the package manually. Also the model could only be trained on a Windows computer (no macOS or Linux), using Firefox (no other browser worked). After resolving these few issues, the package worked nearly flawlessly.

Issues and Improvements: Early on we also decided to make DE NIRO multilingual. This meant training the hotword model in multiple languages, and listening for hotword in all of those languages (other than English we trained Dutch "Hoi DE NIRO", German "Hallo DE NIRO", Spanish "Hola DE NIRO", and Italian "Ciao DE NIRO".

The only concern was running 5 models at the same time made DE NIRO much more sensitive to random speech and noises, somewhat defeating the purpose of using a hotword. In the end we decided to default to English, and only after DE NIRO received the command "Change Language", would it be able to be called with any of the hotwords above, and send the resulting language code

to the rest of the state machine (necessary for speech recognition, interpretation, and text-to-speech conversion).

5.6.2 Speech recognition

Introduction: Done well, speech recognition can be one of the simplest ways to interact with a computational device these days. Provided that commands are intuitive, forgive unclear instructions and are accurately translated into character strings, voice interfaces make devices accessible even to people unfamiliar with them. Thus, choosing speech recognition as the main way of interaction with DE NIRO over a graphical user-interface was the obvious choice for us.
Implementation: Similarly to hotword recognition, we compared three different paths:

1. Offline speech recognition
2. Online speech recognition using an open-source, free "speech rec wrapper package"
3. Using a paid service such as Google Speech Recognition

We discarded offline speech recognition early on in the process, since DE NIRO would realistically always be within range of a WIFI signal or even a smartphone hotspot. Offline packages, we found, are limited in the number of words they accurately translate, and are often less accurate than models in the cloud.

After some trial and error we went with the Freemium Google Speech Recognition. It beat the other online freeware package in perceived performance, and more importantly, it translates speech into text "on the fly", always printing the best current guess to the screen, even if only the first few words of the sentence have been spoken. This was important to us, as (again in the interest of user-friendliness), we wanted the user to receive feedback for interacting with DE NIRO as close to real-time as possible.

Issues and Improvements: Speech recognition never presented any significant issues, and except for the switch from freeware to freemium software which came with an increase in accuracy and speed as described above we didn't require significant improvements here.

5.6.3 Interpretation

Background Parsing out meaning from strings of text is a holy grail of computing, and the domain of natural language processing is one that continues to receive significant volumes of attention in Computer Science and linguistics research. Any user interface led by voice needs to have a route to reliably analyse textual input as well as inform that system architecture of any changes to state. At the minimum, it must be able to find a way to circumnavigate the challenge of (sometimes) incorrect text input to only perform actions that the system is able to.

Research As hinted at Natural Language Processing was to some extent a black box for our group and in particular this sub-team. Within the preliminary research stages and before any implementation decisions were made, we considered both Google's Dialogflow and in particular DeepPavlov, which is an open-source conversational AI library built on TensorFlow and Keras. DeepPavlov has been used to create multiple conversational chatbots, which take a potentially infinite number of string inputs to a finite range of action outputs. This is exactly the technical challenge that our voice interface would have to solve. However, compared to many existing implementations of DeepPavlov within commercial chatbots, we did not have an existing data set of textual inputs (like a search-bar) which match up to action outcomes a user has chosen. This would have required

creating a substantial database of text inputs and corresponding action outputs on which to train our model. Ultimately, we decided would be be outside of the scope of this project.

Implementation We thus decided that the most effective method of implementing speech interpretation would to have a hard-coded set of action items that we would be looking for exact matches for. Given that, in general our voice inputs that required action would almost always be of the form of an action request, sometimes followed by a face and/or object name that is being requested to identify. This involved comparing text input to predefined databases of commands, objects and names that DE NIRO is familiar with (i.e. can identify). This was supported through the highly popular Natural Language Toolkit package supported through python which provided reliable ways to tokenise text input, making the search process considerably easier.

Learnings As had been anticipated, interpretation was a difficult challenge to get working reliably. This was admittedly not helped by a change in approach midway through the project. This could have been avoided with a more substantial initial research period, however given other deadlines and the timeline for reaching MVP stage, it seemed more important to get something functional before we had something that was correct. Whilst not necessarily the most time efficient route to development, the net gains we received as a group for reaching an MVP early on into the project certainly outweighed the extra personal development time within the UI sub-team.

5.6.4 Text-to-speech translation

Introduction: After DE NIRO processes a command and identifies an object or a person, the robot needs to report on what he has seen. Constructing sentences from pieces of information is something that comes natural to humans, but if not carfully considered can result in gibberish such as "I can see a TV monitor, and a TV monitor, and a TV monitor, and a TV monitor" (actual output - shortened by approx. 15 TV monitors - when DE NIRO was asked what it sees in a computer lab with 20 computer monitors.
Implementation: After positive experiences with Google's Speech Recognition service, we decided to implement the corresponding text-to-speech API, as well. The API offers male and female voices for over 20 languages was very easy to implement and works nearly flawlessly. Only names that are not native to the selected language (e.g. Eivinas in English) are pronounced incorrectly.

The true challenge was to construct strings from a wide array of possible inputs that made sense for humans. We broke this task down into four components:

1. **Context** was always decided by the interpreter. Should DE NIRO report on something general (e.g. list all visible objects), specific (e.g. the location of a known person), or request information from the user (e.g. the user's name before learning a new face). A resulting instruction ID provided DE NIRO with this information.

2. **Information type** refers to talking about objects vs. people. In a very simple example "I can see ***a*** tennis racket", is different from "I can see Josh", due to the lack of the corresponding article "a".

3. **Number of variables:** Sentence structure is impacted by the number of objects we refer to. "I can see one person", is different from "I can see one person***, and*** two cell phones", which is different from "I can see one person**,** one tennis racket***, and*** two cellphones". The differences are subtle, but must be accounted for.

4. **Languages** structure sentences in different ways. Whereas English sentences typically assume the structure "Subject, Predicate, Object", German, for example will sometimes pull the object to the front of the sentence.

The solution: We implemented a csv-file-dictionary that would hold text snippets which were strung together by the text-to-speech service, based on the four items above as variables. The resulting output would contain placeholders for the correct names, objects etc. This allowed us to compose grammatically correct sentences, in any language, for any situation DE NIRO was presented with.

5.6.5 Video feedback

Introduction DE NIRO has a full-sized display which has barely been used in previous projects. We thought that in terms of user feedback, however, it could be an invaluable asset. It is often a mystery, why the robot understands what it does, or why it says what it does, and we wanted to make its actions as explainable as possible.

Implementation: We implemented four types of animations helping the user interact with DE NIRO:

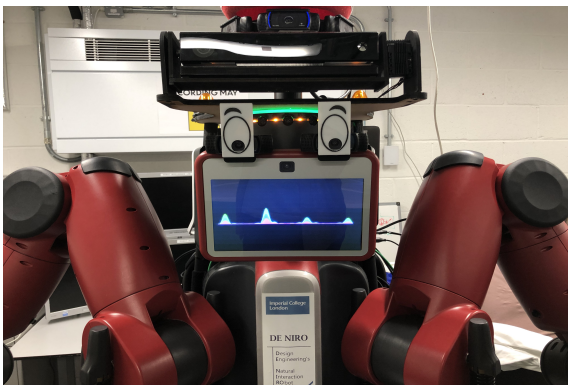
- *Live-speech display:* DE NIRO instantly displays the user's speech input. This helps the user be certain that he or she is being understood.
- *Speech animation:* When DE NIRO speaks, sound-waves appear on the display, similar to Siri's soundwaves on an iPhone. This makes interaction with DE NIRO more natural.
- *Waiting animation:* When DE NIRO waits for input, it displays a slowly moving circle, signaling the user that it is safe to speak. This improves user-experience.
- *Displaying information:* As shown in section 5.2, once an object or a person is recognized, the corresponding output is shown on the screen. Like this, even if extraneous objects are mentioned, the user will be able to identify on the screen, why this was the case.

Issues and Improvements: When sending images to the display, we encountered two issues. First, we tried to develop our own speech-animations, based on the soundwaves DE NIRO was producing. Developing custom animations in real-time, however, proved to be too computationally expensive. Also, lacking a proper design background our animations were nothing compared to those we found on the internet. Thus we recorded videos of animations from different websites, and played those animations back when DE NIRO was speaking. The second problem arose from our solution to the first. The videos were typically longer than the time DE NIRO spent speaking a sentence. To ensure concurrent execution, we ran audio and video in two different threads, but python threads can only be joined, once both threads have terminated. We managed to solve the issue by sending an interrupt from the audio thread to the video thread, once playing back speech had finished, solving the issue.

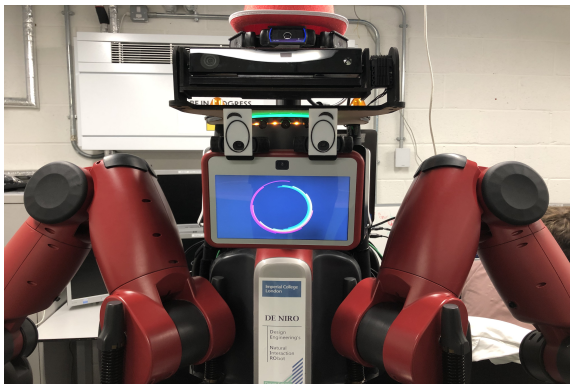
5.7 Movement and pointing

Owners: Nicolas Tobis and Harry Butt

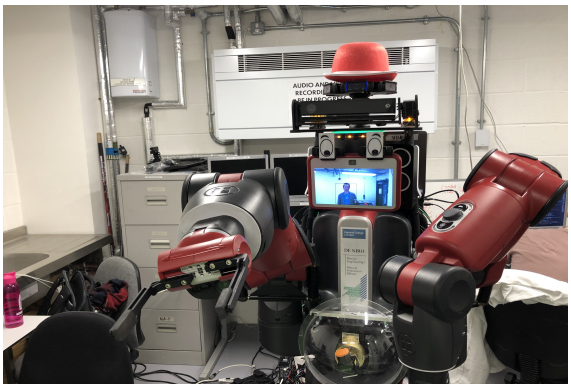
(a) Real-time Speech Recognition Animation



(b) Speech Out Animation



(c) Waiting Animation



(d) Pointing and Recognized Objects in Boxes

Figure 5.11: Possible Live Animations on DE NIRO's Display

The final piece of our demonstration (and getting DE NIRO to truly seem alive) was implementing movement both through its mobile base and Baxter arms.

5.7.1 Movement

Implementation Given previous and ongoing projects which were primarily motivated by movement, as well as the collaborative nature of the Robot Intelligence Lab, we were fortunate enough to be able to mostly re-use an existing code base to support turning on the spot mobile base movement. What was involved in the development process from a time perspective was almost all integration with our system design and hardware challenges. Hardware was, in particular, a constricting factor on this side and a non-functional Inertial Measurement Unit which was replaced led to valuable time being lost to reimplmenting brand new hardware and firmware.

5.7.2 Pointing

Implementation In order to have the robot point to an object that it sees we relied on the ability of the Baxter Robot to record a specific movement. We recorded the left and right arm moving forward, each of which is raised given whether the required object or person stands to the left or right of the robot. If the object is already in front of the robot, the right arm is raised by default.

Further research We believe that work on the arms can be extended substantially. Our initial goal was to have the robot point exactly towards an object, based on where in its frame of reference

in the USB camera the object is detected. Given our time-constraints towards the end of the project, however, we decided to implement a simpler solution that would highlight our core-capabilities, and left room for further research.

6 Final Product

Our project had two primary aims, to expand DE NIRO's computational resources into a remote "Brain Cluster" of CPUs and GPUs and to then use these new resources to expand his capabilities. Our final product achieves these aims by showing how we can augment DE NIRO with remote CPUs running a variety of computationally intensive face, image, object and audio recognition algorithms and bring together the output into a far richer view of the world. Through "Hide and Seek" we demonstrate how this detailed internal model opens up new possibilities for DE NIRO's interactions, such as understanding basic ownership, identification and objects.

There would be little use in having a rich internal model if DE NIRO could only express itself in hardcoded or simplistic routines. Therefore, to take advantage of this expansion of DE NIRO's core capabilities we also developed a flexible and interactive User Interface. This new UI starts with a far more natural intialisation than a mouse click, "Hey DE NIRO". This triggers a state machine that is not a linear set of steps but a never ending loop revolving around generic questions, such as "Who has the X", "Can you see Y", or "Deliver a message". These questions can then be combined and mixed by the user into a large variety of routines or outcomes. In response, we felt it important for the user to see why and how DE NIRO was coming to its output. This led us to heavily use the display in our final product to show what DE NIRO is understanding both from speech recognition and through its new image analysis capabilities.

We feel this new UI, building on the richer model of the environment and enabled by a "Brain Cluster" brings us much closer to DE NIRO being a true "Natural Interaction Robot". We also hope our final product is a really just a first step that highlights the possibility of re-imagining what DE NIRO can do by relaxing his computational constraints.

7 Acknowledgements

We would particularly like to thank our supervisor Petar Kormushev and Roni Permana Saputra who introduced us to DE NIRO and helped us throughout our work.

We also could not have taken on and completed our group project without the tutors and staff in the Department of Computing. We really appreciate all the help we have received both on our group project and throughout our course. Special thanks go to Fidelis Perkonigg for guiding us through, programming, git, testing and so much more.

Appendix A Sprint Phases

Phase 1

1. Study background material (Deadline: 14.01.19)
2. Literature/Package review - Identify state of the art methods for e.g. object recognition or face detection (Deadline: 27.01.19)

Phase 2

1. Build prototype - First standalone (e.g. face-detection on a laptop), then on ROS, using a service (Deadline: 11.02.19)
2. Develop MVP for Robot, i.e. using voice commands iterate through a full cycle of "Who can you see?" and "What can you see?", prompting the robot to provide spoken responses (Deadline: 25.02.19)
3. Enhance DE NIRO's capabilities, e.g. increase situational awareness by giving it the idea of ownership "i.e. which person owns what item" (Deadline: 04.03.19)

Phase 3

1. Add additional features, e.g. display what DE NIRO is "thinking" on the robot display (Deadline: 18.03.19)
2. Fine-tune features, e.g. enhance the update speed of the robot's environment model responsible for its situational awareness (Deadline: 08.04.19)
3. Final System Testing (Deadline: 29.04.19)
4. Finalizing demo and report (Deadline: 15.05.19)

Appendix B Further Research

B.1 Encoding Video and Going Wireless

One avenue for exploration that we see is to use video encoding when sending the video streams to computers running object and face recognition nodes on the "Brain Cluster".

The first reason for this is that network bandwidth is limited and raw video streams are heavy in terms of bandwidth needed. For instance, three 1080p streams at 7 frames per second (FPS) take around 115 MB/s. So if one wants make our system work wirelessly, sending raw video streams from multiple cameras is not viable.

The second reason is that the bandwidth that the laptop on DE NIRO can handle is also limited. The raw video streams worked fine in our case since we only used three cameras at 7 FPS. But we experienced issues with the USB bus getting overloaded when increasing the number of cameras or the number of frames per second.

The solution that we explored is to take the raw video stream from the cameras, encode it on the laptop to H.264 using GStreamer, send the encoded stream to a computer that has a wired connection to the "Brain Cluster", decode the video on that computer and publish the raw video stream for other computers in the "Brain Cluster". This solves the first problem really well since the encoded video is much smaller than the original and can be easily sent over the network (around 1 MB/s for the same configuration of cameras, which is more than 100 times better).

However, this solution does not solve the second problem which is to do with the laptop on DE NIRO that has to take and encode all of these raw video streams and whose USB bus gets overloaded. Fortunately, DE NIRO's multi-camera rig is composed of many *Logitech C920* cameras, which support on-device encoding to H.264 [19]. This means that it is possible to request H.264 video directly from the device and not overload the USB bus. We also predict that such solution would produce less lag because encoding video on the laptop probably takes a bit more time than encoding on the camera itself that has dedicated hardware for this task. So the object and face recognition nodes would get the images from the cameras quicker.

Due to time limitations and our particular three camera setup we simply went for sending the raw video over an Ethernet cable. But we see encoding the video stream as an exciting opportunity to make DE NIRO's Brain Cluster go wireless.

Appendix C Project Management

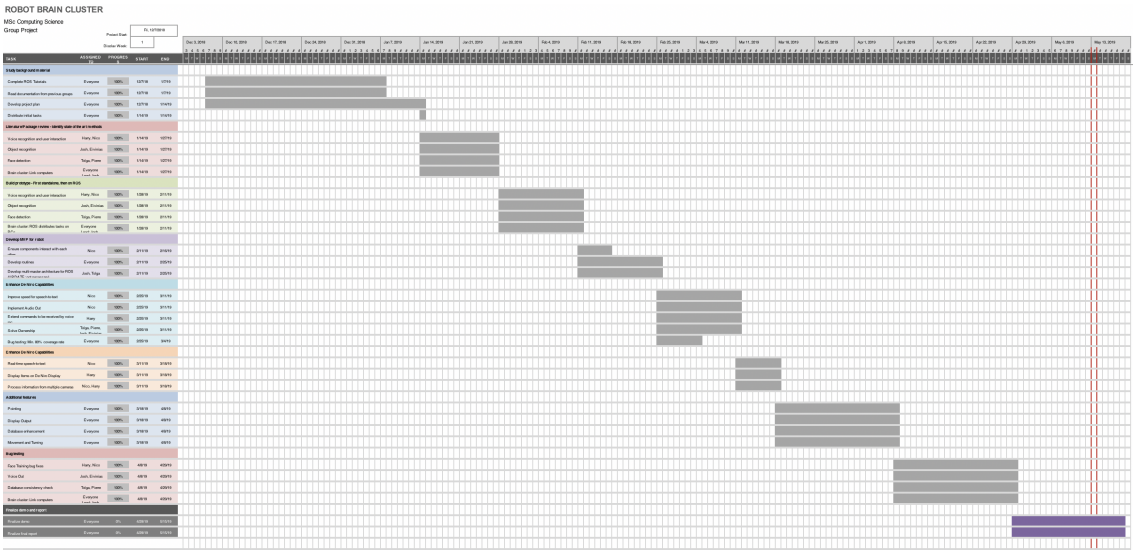


Figure C.1: Overall Project GANTT Chart

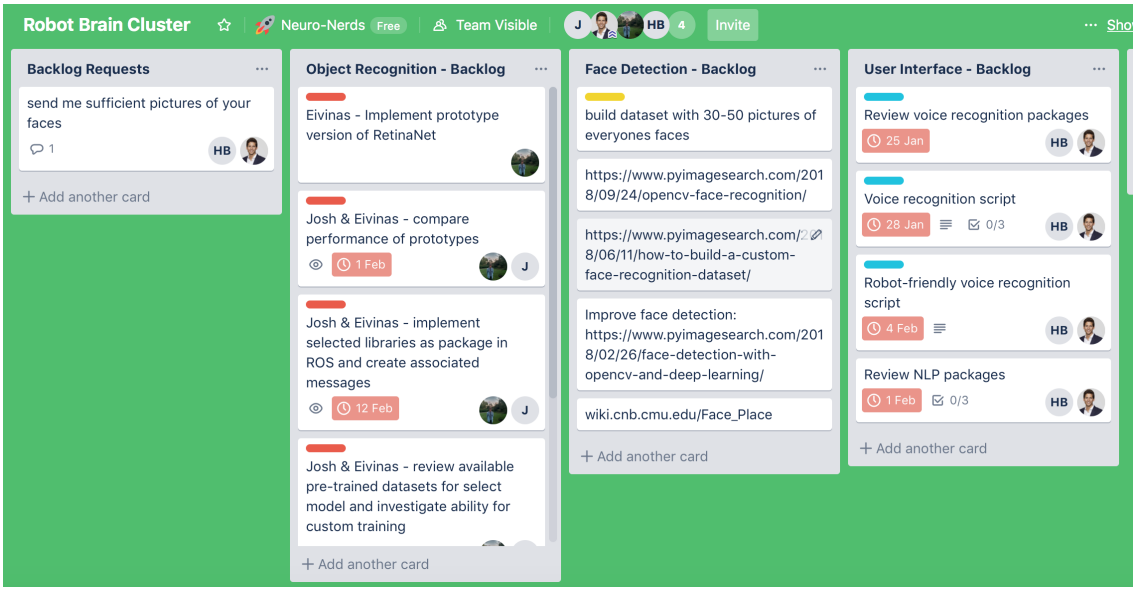


Figure C.2: Early Use of Trello

Appendix D Unit Testing

```
# Create ROS Functionality for Python
def kill_child_processes(parent_pid, sig=signal.SIGTERM):
    try:
        parent = psutil.Process(parent_pid)
        print(parent)
    except psutil.NoSuchProcess:
        print("parent process not existing")
        return
    children = parent.children(recursive=True)
    print(children)
    for process in children:
        print("try to kill child: " + str(process))
        process.send_signal(sig)


class Roscore(object):
    """
    roscore wrapped into a subprocess.
    Singleton implementation prevents from creating more than one instance.
    """
    __initialized = False
    def __init__(self):
        if Roscore.__initialized:
            raise Exception("You can't create more than 1 instance of Roscore.")
        Roscore.__initialized = True
    def run(self):
        try:
            self.roscore_process = subprocess.Popen(['roscore'])
            self.roscore_pid = self.roscore_process.pid  # pid of the roscore process
    (which has child processes)
        except OSError as e:
            sys.stderr.write('roscore could not be run')
            raise e
    def terminate(self):
        print("try to kill child pids of roscore pid: " + str(self.roscore_pid))
        kill_child_processes(self.roscore_pid)
        self.roscore_process.terminate()
        self.roscore_process.wait()  # important to prevent from zombie process
        Roscore.__initialized = False
```

Figure D.1: Roscore Implementation for Unit-Tests

Module ↓	statements	missing	excluded	branches	partial	coverage
src/FaceRecognition.py	23	0	0	0	0	100%
src/ModelTrainer.py	76	1	0	12	2	97%
src/VideoRecognition.py	118	9	0	24	4	89%
src/__init__.py	0	0	0	0	0	100%
Total	**217**	**10**	**0**	**36**	**6**	**93%**

coverage.py v4.5.2, created at 2019-03-03 19:56

Figure D.2: Code Coverage Screenshot for the Face Recognition Package

Module ↓	statements	missing	excluded	coverage
yolo_publisher.py	52	7	0	87%
Total	**52**	**7**	**0**	**87%**

coverage.py v4.5.2, created at 2019-03-02 15:43

Figure D.3: Code Statement Coverage for the Object Detection Package

Module ↓	statements	missing	excluded	branches	partial	coverage
yolo_publisher.py	52	7	0	4	1	86%
Total	**52**	**7**	**0**	**4**	**1**	**86%**

coverage.py v4.5.2, created at 2019-03-02 19:28

Figure D.4: Code Branch Coverage for the Object Detection Package

Module ↓	statements	missing	excluded	coverage
brain_cluster_database.py	90	0	0	100%
Total	**90**	**0**	**0**	**100%**

Figure D.5: Statement Coverage for Brain Cluster Database

Module ↓	statements	missing	excluded	branches	partial	coverage
brain_cluster_database.py	90	0	0	42	1	99%
Total	**90**	**0**	**0**	**42**	**1**	**99%**

Figure D.6: Branch Coverage for Brain Cluster Database

Module ↓	statements	missing	excluded	branches	partial	coverage
hotword_client.py	14	0	0	0	0	100%
hotword_service.py	27	0	8	2	0	100%
Total	**41**	**0**	**8**	**2**	**0**	**100%**

coverage.py v4.5.2, created at 2019-03-03 15:14

Figure D.7: Code Branch Coverage for the Hotword Recognition Package

Module ↓	statements	missing	excluded	branches	partial	coverage
__init__.py	0	0	0	0	0	100%
speech_rec_client.py	14	0	0	0	0	100%
speech_rec_service.py	36	0	8	8	1	98%
Total	**50**	**0**	**8**	**8**	**1**	**98%**

coverage.py v4.5.2, created at 2019-03-03 15:33

Figure D.8: Code Branch Coverage for the Speech Recognition Package

Module ↓	statements	missing	excluded	coverage
__init__.py	0	0	0	100%
interpreter_class.py	86	4	0	95%
interpreter_client.py	13	2	0	85%
interpreter_service.py	11	0	6	100%
test_interpreter.py	72	0	32	100%
Total	**182**	**6**	**38**	**97%**

coverage.py v4.5.1, created at 2019-03-04 11:13

Figure D.9: Code Branch Coverage for the Command Translation Package

Module ↓	statements	missing	excluded	coverage
text_to_speech.py	144	2	0	99%
Total	**144**	**2**	**0**	**99%**

Figure D.10: Code Branch Coverage for the Text To Speech Package

Module ↓	*statements*	*missing*	*excluded*	*coverage*
security_check.py	23	0	0	100%
Total	**23**	**0**	**0**	**100%**

Figure D.11: Code Branch Coverage for the Security Check Package

Appendix E COCO Dataset YOLOv2 is Trained On

person	elephant	wine glass	diningtable
bicycle	bear	cup	toilet
car	zebra	fork	tvmonitor
motorbike	giraffe	knife	laptop
aeroplane	backpack	spoon	mouse
bus	umbrella	bowl	remote
train	handbag	banana	keyboard
truck	tie	apple	cell phone
boat	suitcase	sandwich	microwave
traffic light	frisbee	orange	oven
fire hydrant	skis	broccoli	toaster
stop sign	snowboard	carrot	sink
parking meter	sports ball	hot dog	refrigerator
bench	kite	pizza	book
bird	baseball bat	donut	clock
cat	baseball glove	cake	vase
dog	skateboard	chair	scissors
horse	surfboard	sofa	teddy bear
sheep	tennis racket	pottedplant	hair drier
cow	bottle	bed	toothbrush

Table 1: 80 Common Objects in COCO Dataset

Appendix F Time Log - Estimates

Time Allocation

Tasks	Josh	Tolga	Nico	Harry	Eivinas	Pierre	Total % of Project Time
Brain Cluster	25%	15%	15%	15%	15%	15%	10%
Face Recognition		50%				50%	7.5%
Face Training		50%				50%	7.5%
Spoofing Detection		50%				50%	5%
Generic Object Recognition	75%				25%		5%
Particular Object Recognition	10%				90%		5%
Environment Model	50%				50%		10%
State Machine			100%				10%
Audio In				100%			5%
Audio Out			100%				2.5%
Display			100%				2.5%
Movement			25%	75%			5%
Admin and Organisation	5%	5%	30%	50%	5%	5%	5%
Reports	15%	15%	15%	25%	15%	15%	20%

Appendix G Project Log - Minutes

Please view a complete record of our project logs and minutes at:

https://drive.google.com/drive/folders/11ijx6eRxmsNjHXdTGnzwTwpw-suXym8z

References

[1] Robot Intelligence Lab
http://www.imperial.ac.uk/robot-intelligence/

[2] Petar Kormushev's website
http://kormushev.com/

[3] Robot DE NIRO
http://www.imperial.ac.uk/robot-intelligence/robots/robot_de_niro/

[4] Software Engineering Economics
https://csse.usc.edu/TECHRPTS/1984/usccse84-500/usccse84-500s.pdf

[5] Autonomous Snack Delivery Android (ASDA)
http://nikonikolov.com/projects/asda

[6] Project Fezzik: Playing Fetch with Robot DE NIRO
https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/1718-pg-projects/Group7-Play-Fetch-with-Robot-DE-NIRO.pdf

https://robot-intelligence-lab.gitlab.io/fezzik-docs/index.html

[7] Robot Operating System
http://www.ros.org/

[8] YOLO: Real-Time Object Detection
https://pjreddie.com/darknet/yolo/

[9] YOLOv2 Object Recognition (darkflow implementation)
https://github.com/thtrieu/darkflow

[10] COCO: Common Objects in Context dataset
http://cocodataset.org

[11] Imutils Library
https://github.com/jrosebr1/imutils

[12] OpenCV Library
https://opencv.org

[13] PyImageSearch: Website With Many Useful Computer Vision Tutorials
https://www.pyimagesearch.com/

[14] Dlib Library
http://dlib.net

[15] Deep Learning Haar Cascade Explained
http://www.willberger.org/cascade-haar-explained/

[16] Rapid Object Detection using a Boosted Cascade of Simple Features Paul Viola and Michael Jones 2001

[17] find_object_2d ROS package
http://wiki.ros.org/find_object_2d

[18] Wikipedia: Scale-invariant feature transform
https://en.wikipedia.org/wiki/Scale-invariant_feature_transform#Comparison_of_SIFT_features_with_other_local_features

[19] Using the Logitech C920 webcam with GStreamer
http://oz9aec.net/software/gstreamer/using-the-logitech-c920-webcam-with-gstreamer