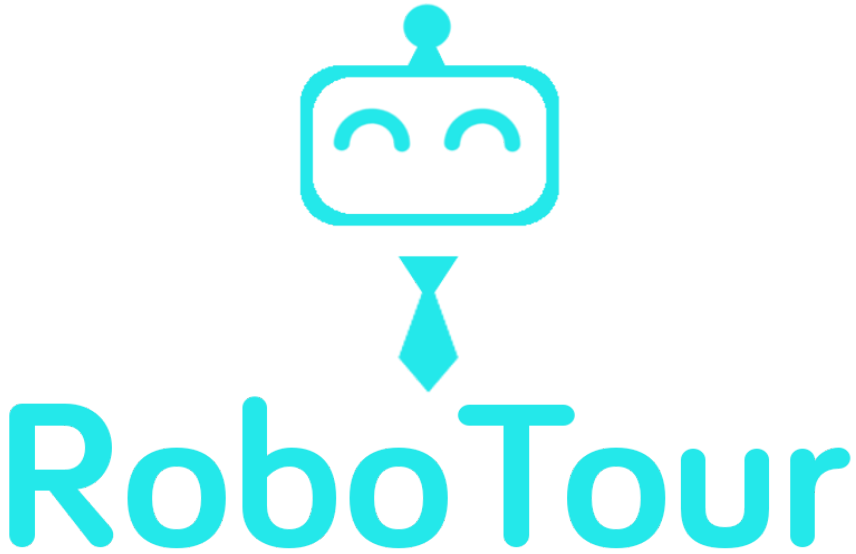
# System Design Project

## Technical Report




**Group 18**
Alice Wu
David Speers
Deividas Lavrik
Finn Zhan Chen
Mahbub Iftekhar
Michal Dauenhauer
Mariyana Cholakova

# 1 Introduction

RoboTour is a robotic tour guide that assists people in environments such as museums or art galleries. The system comprises an autonomous robotic guide, a purpose-built Android application, and a web server mediating the communication between the two. RoboTour can be controlled by up to two Android devices, and the tour may be followed by many more devices. The app allows users to interact with RoboTour intuitively in multiple languages.

# 2 Components

## 2.1 Robot Hardware Components

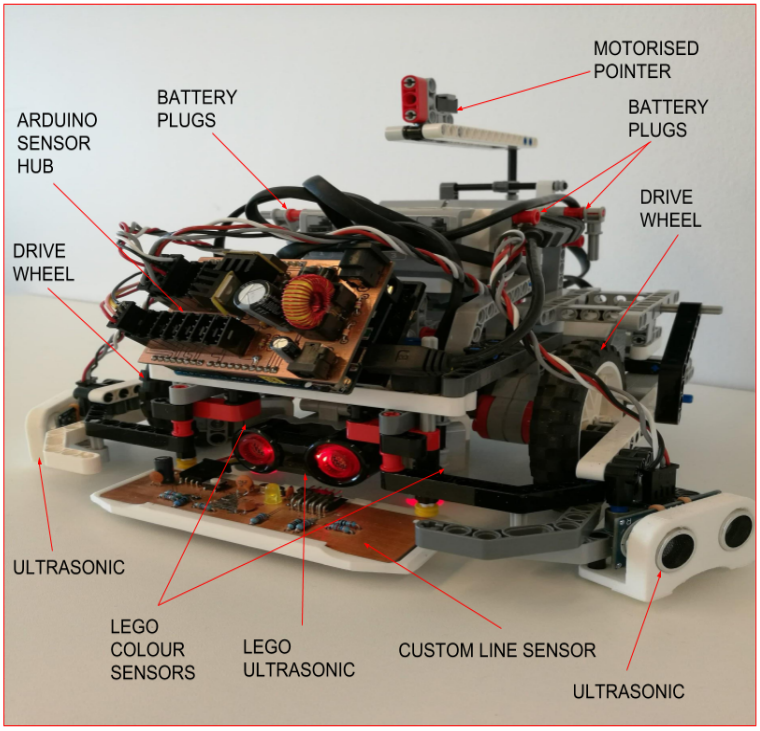



**Fig. 1:** Labelled view of RoboTour

**Table 1:** Robot Components

| Quantity | Item |
|---|---|
| 2 | HC-SR04 Ultrasonic Sensors |
| 1 | LEGO Ultrasonic Sensor |
| 2 | LEGO colour Sensors |
| 1 | Custom Line Sensor |
| 1 | Motorised Pointer |
| 2 | Drive Wheels |
| 1 | Arduino Sensor Hub |
| 1 | Lego EV3 |

The robot (See Fig. 1) is a differential drive platform, i.e. the movement is achieved with two motorised drive wheels. Varying the rotational speed of the wheels independently, allowed us to introduce rotation of the chassis in addition to the linear translation. Additionally, two rear wheels are added for stability and weight support. They were designed with the aim of minimising the friction and disturbance to the robot control.

The differential drive platform was chosen due to the mechanical simplicity and intuitive control. Using two motors means that the robot can exercise control over only two degrees of freedom out of three on the plane. This means it lacks the ability to move in any direction besides the one it is facing. However, the envisioned primary mode of operation (following predetermined paths) can be fulfilled without this kind of movement.

The chassis of the robot was designed to be rigid and compact. Rigidness eliminates additional variables and random disturbances (e.g. occasional misalignment of the wheels due to load). The small size of the robot was essential in accomplishing obstacle avoidance in the small test area.

The robot also features a motorised pointer which is used during tour-guiding to engage users and provide a visual hint about the location of the painting. The pointer is driven by a worm gear, whose self-locking properties prevent the pointer from being accidentally misaligned. The pointer is mounted above the other components of the robot so that it is visible to the users. It rotates in the plane parallel to the floor.

The sensor suite of the robot includes three ultrasonic sensors for obstacle detection and avoidance, two LEGO colour sensors for detection of markings on the floor, and a custom line sensor to track the lines used for navigation. All three ultrasonic sensors were placed low above the ground, as the initial tests indicated that some obstacles (e.g. human shoes) are better detected with such placement of the sensors. Additionally, since both the LEGO colour sensors and the custom line sensor operate by detecting the intensity of the light reflected from the floor, their precise placement was crucial to the reliability of the robot. The mounting of the sensors allows for precise adjustments of the distance to the ground.

As the required number of sensors exceeds the capabilities of the stock EV3 set, we decided to use an Arduino board as a sensor hub, which allowed us to connect the HC-SR04 sonars as well as the custom sensor (see Fig. 2).

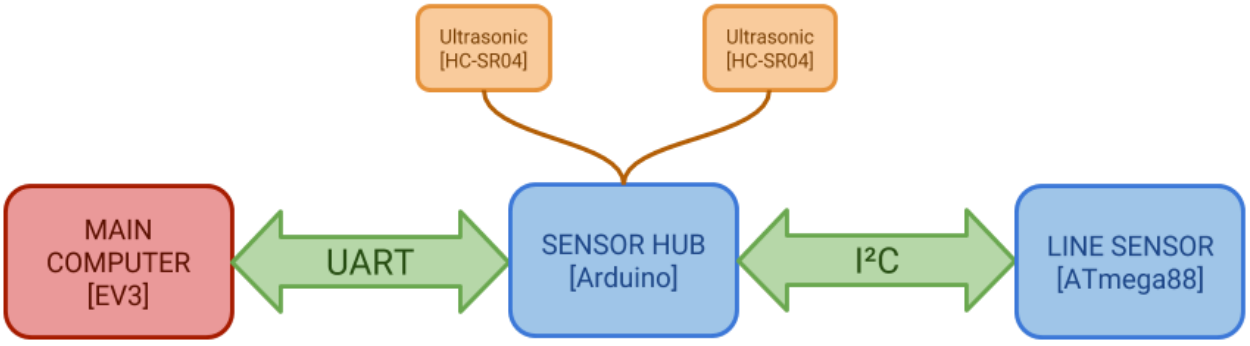



**Fig. 2**: Sensor Expansion Schematic

### 2.1.1 Sensors

**Table 2**: Sensor Specifications

| Sensor | Principle of operation | Returned value | Purpose |
|---|---|---|---|
| **LEGO Ultrasonic Sensor** | Time-of-flight distance measurement | 0-2550 [mm] | Obstacle detection |
| **HC-SR04 Ultrasonic sensor** | Time-of-flight distance measurement | 0-2550 [mm] | Obstacle avoidance |
| **LEGO Colour sensor** | Colour detection based on RGB values of light reflected from the object | 0-7 (Colour code) | Branch markers detection |
| **Custom line sensor** | Line position based on 6 line detectors (infrared reflectance transducers) | 10-60 (Line centre position)<br>6xBool (detector activation) | Line following<br>Branch detection |
| **Drive wheel tachometers** | Rotation measurement based on In-built quadrature encoder | Signed Integer [deg] (motor rotation) | Odometry |
| **Pointer tachometer** | Rotation measurement based on In-built quadrature encoder | Signed Integer [deg] (motor rotation) | Pointing to art pieces |

### 2.1.2 Sensor Hub

The sensor hub is an Arduino Uno board with a custom shield featuring ultrasonic sensor connectors as well as access to the I2C bus. The hub is responsible for three functions:

- Communication with the custom sensor
- Processing of the signals from the side sensors
- Communication with the EV3

The firmware is written in ArduC, using standard Arduino libraries. The communication with the custom sensor takes place over I2C with the Arduino operating in master mode. In each loop, the hub requests 6 bytes (one per each detector) from the line sensor and updates its inner buffer. The HC-SR04 ultrasonic sensors are handled using a NewPing Library. The sensor routine measures the time it takes for an ultrasonic wave to reach the obstacle and bounce back. The time is then used to calculate distance, and the result is stored in the buffer.

The communication with the EV3 happens over UART (serial port). UART was chosen over I2C due to its ease of access to the serial compared to I2C and because it frees a USB port on the EV3. The EV3 can send a request (a single byte message) to the Arduino, which responds by transmitting the cached sensors readings. Initially (V1.0), the data was transmitted in a frame formed with ASCII characters and parsed by the EV3. This, however, resulted in a polling rate too low for reliable and smooth operation. In the next version (V2.0) we opted for transmitting the values in pure numerical form (i.e. each byte stored the binary value of the corresponding sensor). This resulted in a reduction of polling time by a factor of 3. Further refinement (V2.1) involved reducing debugging and diagnostic features on the EV3 side, resulting in another decrease in polling time (See Fig. 3).

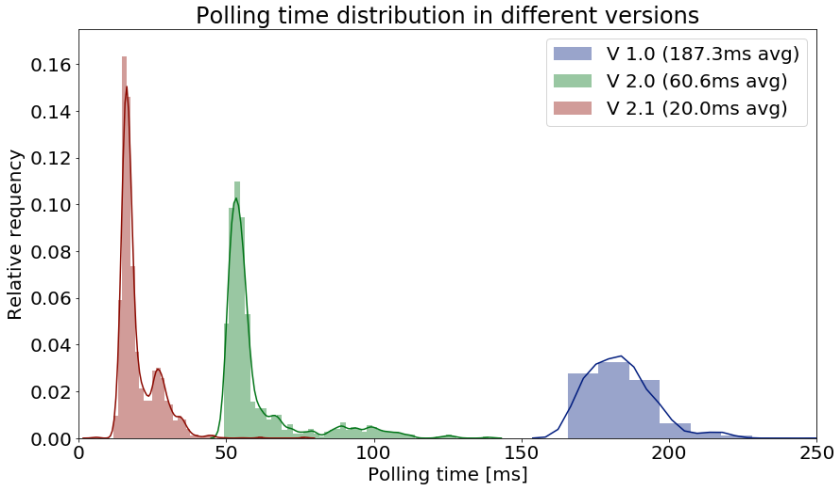



**Fig. 3**: Polling time distribution

### 2.1.2 Custom Sensor

The custom sensor was designed to meet the following objectives:

- High tracking range - the stock LEGO colour sensor allows for line following by tracking the edge of the line. This, however, means that there is approximately a 10mm region in which the sensor readings show useful variation. This makes it necessary to make sharp corrections to stay in that region. Expanding it makes the robot more resilient to small disturbances and appear more natural.
- High effective resolution - by using two LEGO sensors far apart it is possible to extend the tracking range. However, the region between them is “dead” (no meaningful readings are made). It is important that the value changes gradually so that more sophisticated control can be implemented resulting in smoother operation.

- High polling rate - a high refresh rate allows the robot to run the control loop at higher frequencies and react quickly enough even at higher speeds. It also ensures that the branches are not missed.
- Differentiation between sides - in a case when there is no line detected, it should be possible to determine where to look for the line. This way the penalty for overshooting the line is minimised.

To achieve these goals, the sensor was designed as an array of 6 QRE1113 infrared reflective sensors connected to an ATMega88A microcontroller. Each sensor is an IR LED and phototransistor in a single package. The output voltage depends on the amount of light that reaches the phototransistor after reflecting from the surface - dark surfaces absorb light resulting in high readings, while bright surfaces reflect light resulting in low readings.

The firmware for the ATMega88A was written in C using AVR Toolchain (Microchip, 2017). The microcontroller uses an Analogue to Digital Converter (ADC) to convert the voltage level to an 8-bit number for further processing. The readings are taken and cached continuously using an interrupt-driven method to maximise the reading rate. The sensor joins the I2C bus as a slave - on the address match, the sensor begins the transmission of the cached sensor readings. Additionally, a single LED is added as a status indicator.

The sensor schematic and PCB were designed using Autodesk Eagle. The two-sided board was CNC-milled and populated and soldered by hand. A housing compatible with LEGO was designed in Autodesk Inventor and 3D printed on Ultimaker3 available in UCreate Studio for easy integration with the robot. The final version of the complete sensor can be seen in Fig. 4a & 4b.

The EV3 receives raw data - six digitised analogue readings. To account for variances between parts and external conditions (such as light in the room or tape degradation), each run of the robot should begin with the calibration of the sensor. While rotating around its axis, the robot records minimum and maximum readings for each of the six detectors separately. Then thresholds are calculated which are used to determine whether a given detector is over the line (i.e. is activated). Each of the detectors is assigned a position value (see Fig. 4b). The position of the line (sensor output value) is calculated as an average position value of all activated detectors.

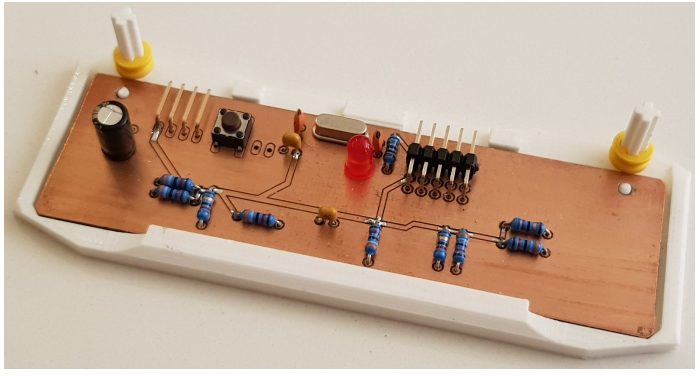


**Fig. 4a**: Top view of the custom sensor in the 3D-printed housing

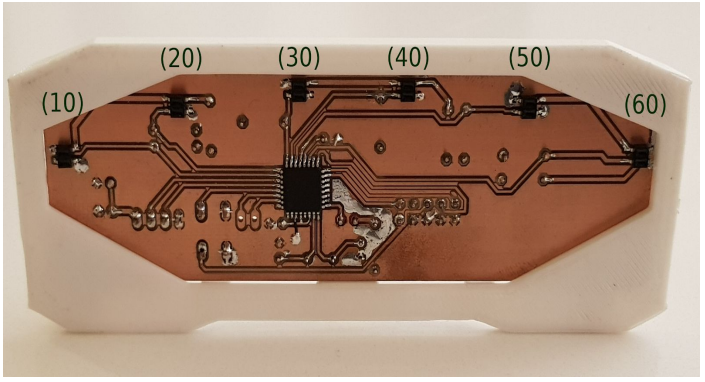



**Fig. 4b**: Bottom view of the Custom sensor with the position values indicated above the detectors

## 2.2 Robot Software Components

### 2.2.1 Programming Language

The main computing node of the robot is the LEGO EV3 running an ev3dev system.
The system provides easy and robust access to the hardware with some high-level functions. There are some available bindings, including ones in Java, Python or C. We opted for Python due to a number of its features:

- Clarity of the code and self-documentation.
- Easy development and deployment due to interpreted nature of the language.
- Availability of easy-to-use modules such as Threading or Serial.
- Flexibility in using elements of Object-Oriented, Functional and Procedural programming.

While the performance of the code could be increased by using C, the need to cross-compile the code would make the deployment more complex and less intuitive, while requiring additional support infrastructure (such as setting up the proper toolchain).

Alternative computing platforms were considered (e.g. Raspberry Pi 3). However, the EV3 platform had the advantage of providing a well-functioning native hardware interface including sensor drivers and motor controllers supporting closed-loop position and speed control. Gains from the additional computing power would not outweigh the need to research and develop analogous solutions for the different platforms.

### 2.2.2 Robot program architecture

We developed a Finite State Machine-based framework that the main program was based on. It was designed to provide some abstractions to separate different aspects of the robot operation.

In each loop of the main program, all sensor readings are performed and stored in an Environment object, so that the values are consistent across the entire loop. Then, the FSM makes transitions based on the current state of the environment if necessary. Finally, a dispatcher object is used to execute an action or a single step of an algorithm associated with the current state of the FSM. This design follows the “sense-plan-act” principle.

Finite state machines were used, as it seems natural to think of the robot as being in different modes, or states, and behaving appropriately to that mode/state (e.g. line-following mode, obstacle avoidance mode). Additionally, the FSM approach allows separating the program flow control from the algorithms themselves. In turn, this simplifies adding additional sensor processing or modifying control logic, as they are separated from one another and seemingly run concurrently (e.g. even if the robot waits, it still updates the sensor readings). The flow of the main loop can be seen in Fig. 5a while the simplified state diagram is presented in Fig. 5b. The additional strength of the FSM approach is the ability to apply it at the different levels of abstraction, i.e. each algorithm can also be run by an FSM, while being triggered by the master FSM.

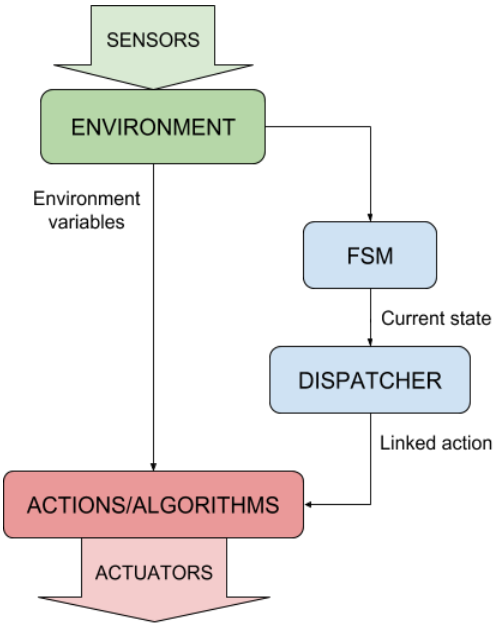



**Fig. 5a**: Robot Control Structure

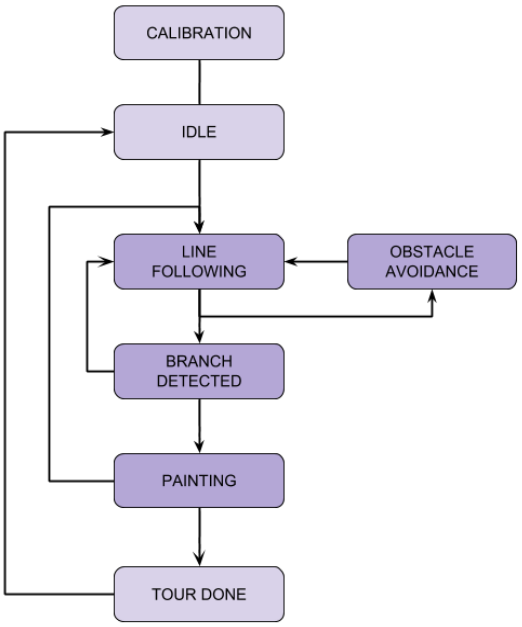



**Fig. 5b**: Simplified Master FSM Diagram

### 2.2.3 Line Following

Some of RoboTour's essential features are reliable position estimation and navigation. To achieve this, many different alternatives were considered, such as:

- **RFID tags at important locations** - which would be detected by RoboTour as it moved around the museum. This option would have been cheap, but RFID tags have a very limited range, only tell an approximate position, and do not provide the orientation of the robot.
- **Overhead camera** - very effective for estimating the robot's position, orientation and finding the optimal path around obstacles. However, in real museums the cost of installation of multiple high-quality cameras on the ceiling could be very difficult as some buildings are listed and modifying them to implement such a solution would introduce risks and complications. In the end, we dropped this idea due to not being convinced we would successfully implement it in the given time frame and skillset of the group - especially as we knew we would possibly be a member down for a while, which turned out to be the entire semester. Thus looking back at this, we believe this decision was correct considering our circumstances.
- **Odometry** - unreliable as any error that occurs during a tour accumulates over time, giving large mismatches between estimated and actual positions.

In the end, a decision was made to go with line following (with short lines placed perpendicularly to the path to indicate branches and paintings), as we already had the tools (LEGO colour sensor/light reflectance sensors) and experience implementing such a solution. This idea was also attractive because a properly done line following robot always sticks to a carefully pre-laid path and does not need to do heavy computations by itself.

The line following is done using a Proportional-Integral-Derivative Controller (PID). The main benefit of using PID instead of other simpler approaches is reliability and reduced shakiness when following the line. To help better illustrate the reason for our chosen algorithm, Fig. 6a and 6b illustrate how the PID controller compares with a boolean line follower.

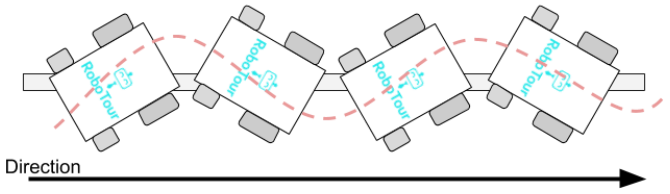



**Fig**. 6a: Line Following **Without** PID

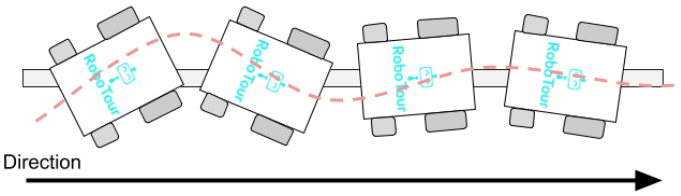



**Fig**. 6b: Line Following **With** PID

### The workings of PID

The following paragraphs explain each of the three parts of the PID algorithm.

The proportional part is responsible for turning the robot at a speed proportional to the error between the target (expected) value and custom line sensor readings. This is a basic driving force, which makes the robot turn faster as the error gets bigger.

The integral part sums up an error from the past loops of the program and reduces or strengthens the turning rate proportionally to the size and sign of the error history. This allows the robot to correct its movement faster when following curvy parts of the line.

The derivative part takes the current turning rate, multiplies it by a certain constant and applies it as an input to the motors. This is done to dampen the oscillations which can arise when using a PI controller. The reduced wiggliness helps the robot reach its equilibrium position and follow the line smoothly.

## 2.2.4 Environment

The museum environment is imported into the robot as a graph implementation method, and this consists of 3 hash maps.

The first map is a dictionary that represents the distance between each neighbouring vertices (branch or painting). This is used to calculate the shortest path to the closest painting using Dijkstra's algorithm.

A second map is a dictionary which consists of vertices (branch or painting) and the orientation of its neighbouring vertices. This is used for aligning the robot with the desired direction and follow the line to the desired branch.

The third map represents the position of the paintings in the map. This is used for deciding where the pointer should turn to point to the desired painting.

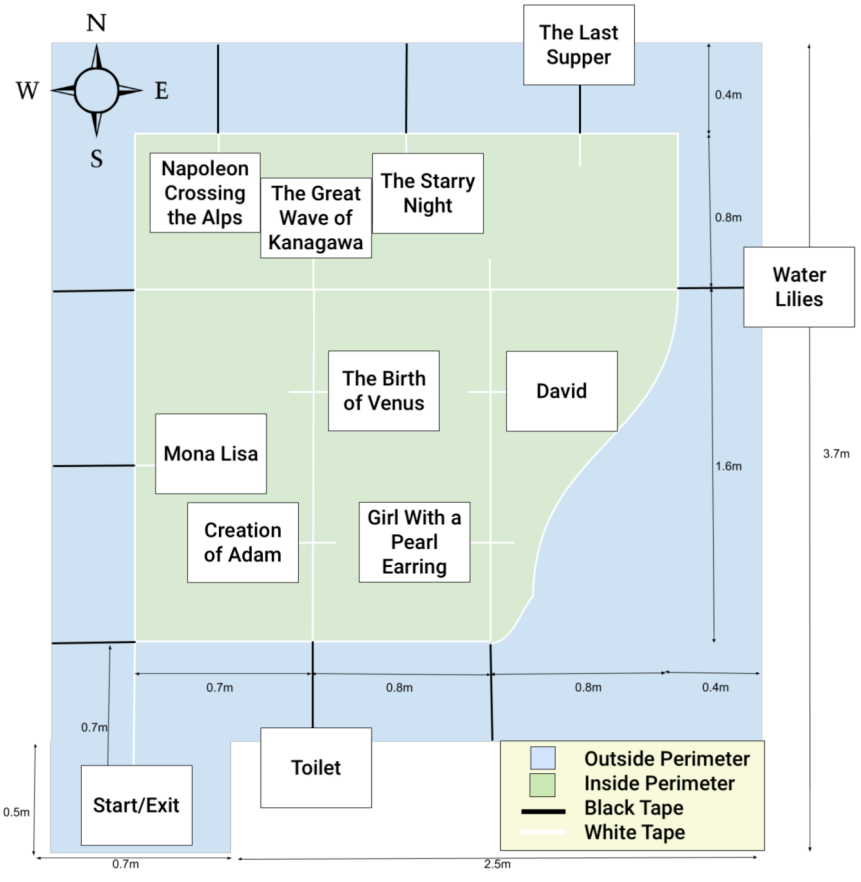



**Fig. 7:** Map of the Museum Environment

### 2.2.5 Route Planning Algorithm

Once the robot has received the paintings selected by the user, the best route is planned. The map of the museum (Fig. 7) is imported as a graph (using a python dictionary) with vertices representing branches or paintings and with edges representing the action required to move to the neighbouring vertex. The graph implementation method enables Dijkstra's algorithm (Skiena, 1990) to find the closest painting in the graph (using the distance cost function).

We decided to represent the environment as a graph, as it enables us to use some of the most common path finding algorithms. Alternatives to Dijkstra's graph search algorithms were considered such as A* search algorithm and the Viterbi algorithm, which are faster than Dijkstra's algorithm when there are a large number of vertices. However, they require a more complex implementation, and because in our prototype there is a very small amount of vertices the speed difference would've been negligible, so we chose Dijkstra. Furthermore, we found existing solutions for Dijkstra's algorithm, and we tailored it to suit our graph implementation of the map (Gilles-bertrand.com, 2018).

Dijkstra's algorithm is run with the robot's position and the selected stack of paintings (For example, Painting A, and Painting B). The closest painting (Painting A) in the stack and its path (an ordered list of branches or paintings) to reach it are returned. Then the robot aligns itself to the desired direction and line following is executed. When the path is finished, Painting A is removed from the stack, and robot's position is updated to Painting A so that Dijkstra's algorithm can be run with the new position and the stack to find the next closest painting (Painting B). This repeats until the stack is empty. Then the order of the paintings is uploaded to the server so that the Android app can update its UI.

### 2.2.6 Obstacle Avoidance Algorithm

Considering the line may be blocked, we have implemented obstacle avoidance to prevent wasting the user's time from waiting for the obstacle to be removed.
The front LEGO ultrasonic sensor is used to detect the obstacles. If there is an object within 20cm of the front of the robot, it is considered an obstacle and the robot will halt its movement and attempt obstacle avoidance.

### 2.2.7 Wall Following Mode

Considering the obstacle may have various sharp edges, one of the best ideas for the robot to avoid the obstacle is to keep a constant distance from the obstacle and move along the sharp edge. This is shown in Fig. 8a.

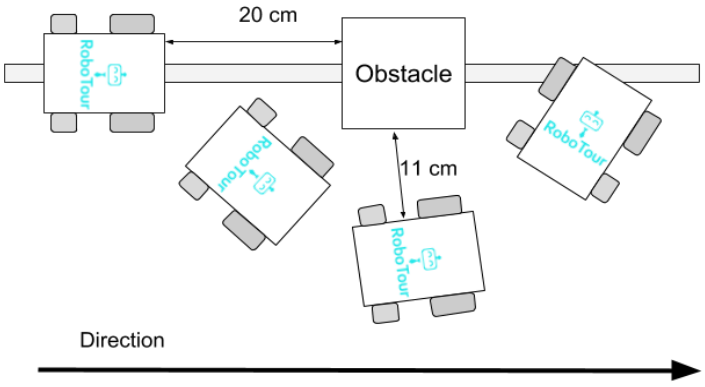



**Fig. 8a**: Obstacle Avoidance

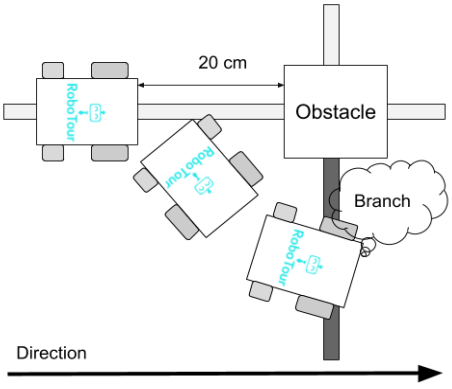



**Fig. 8b**: Black Line Detected

When an obstacle is detected, the robot will turn 90 degrees in the direction that is pre-programmed in the dictionary (See 2.2.4). The two HC-SR04 ultrasonic sensor at the side are used to detect the distance

to the obstacle. The robot will keep the distance at 11 centimetres, by adjusting itself based on the distance it detected at the side ultrasonic sensor. When the white line is detected by the custom line sensor, the robot will turn 90 degrees again and go back to line following mode.

**Note**: If there is not enough space for the robot to pass through, it will stop and notify the user in the app in their selected language.

### 2.2.8 Obstacle on Branch

As the robot counts branches to understand where it is in the environment, when an obstacle is on a branch, the robot may miss the branch during the obstacle avoidance, which will make it lose its place. Black lines have been introduced at the branch to solve this issue. The two LEGO colour sensors are used to detect the colour of the tapes. In obstacle avoidance mode, if the black line is discovered - when the LEGO sensors give a reading of 1 - the robot knows it has missed a branch (Fig. 8b). If the branch is a painting it needs to go to, it will turn and wait in front of the obstacle until the obstacle is removed. Otherwise it will continue obstacle avoidance and count for a branch.

### 2.2.9 Indicating Paintings

The pointer is used to indicate the position of the paintings. Painting position at each branch is pre-programed into the code. When the robot arrives at the painting, based on its position and orientation, it can calculate how many degrees the pointer should turn to indicate the painting. When the user presses “continue” on the app the robot will return the pointer to the default position and start navigating to the next painting.

## 2.3 Server Communication

The communication between the EV3 and all Android phones are done via a server hosting a PHP file. The reason for choosing this over other mediums - such as a direct Bluetooth connection - is due to the limitations of direct Bluetooth connection with an Android phone. We found that a direct Bluetooth connection is not as reliable, and also that it limits it to one device to be connected to the EV3, making multi phone support not feasible. Due to these reasons, we decided to go for a server to mediate messages between the Android apps and EV3.

The EV3 connects to the server via the internet. It gets an internet connection from Bluetooth tethering. The reason we chose this over WiFI with a dongle is due to the unreliability of the conman WiFi driver. We discovered that once the WiFi driver was set up, the connection speed nearly doubled compared to Bluetooth Tethering. However, the WiFi would disconnect intermittently despite all necessary certificates being installed due to the conman WiFi driver. We decided reliability was more important for our use case hence we chose Bluetooth tethering.

## 2.4 App

According to The Verge, 81.7% of smartphones in the market are running Android in 2016 (The Verge, 2018). This is why an Android app was developed to be the user interface for the user to communicate with the robot. We believed that due to the time restrictions, we should focus on developing a prototype for only one platform.

### 2.4.1 App Development

We have made the app backwards compatible with older versions of Android; the app will work with Android SDK version 17 onwards (users also require 20mb free space and an internet connection to download and use it). The app was developed in Android Studio 3.1 using Kotlin. We choose to develop in Kotlin instead of the more common Java as we had experience in Android development using Kotlin and

the null safety that Kotlin provides as standard ensures the app can be made more robust with less effort as compared to Java. Kotlin also ensured the app has approximately 20% less code than that would be required with Java, making the app easier to debug and adapt.

### 2.4.2 User Interface


**Fig. 9a:** MainActivity.kt


**Fig. 9b:** SelectLanguageActivity.kt

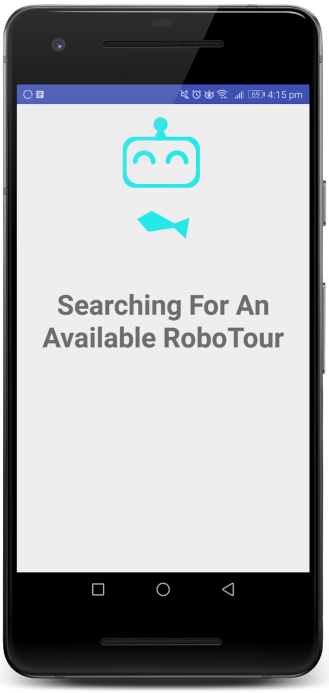



**Fig. 9c:** WaitingActivity.kt

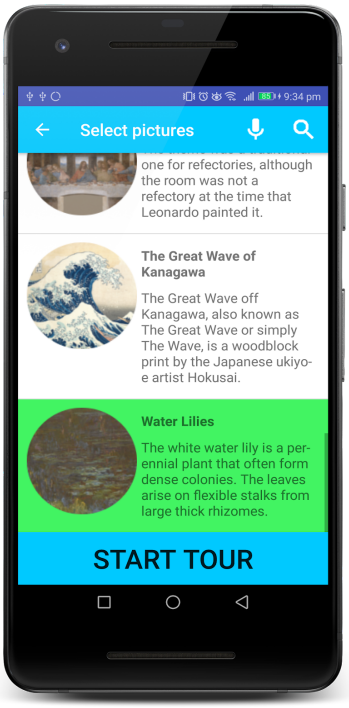



**Fig. 9d:** SelectPicturesActivity.kt

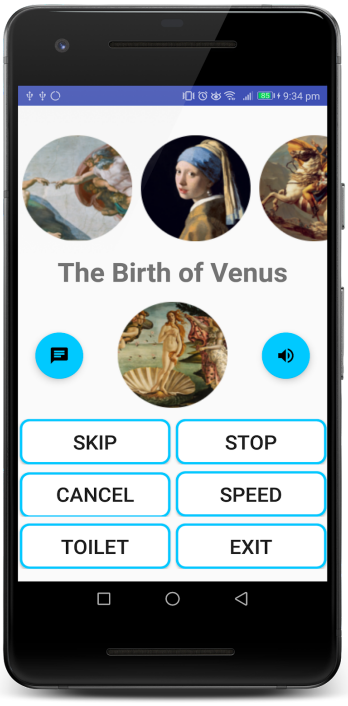



**Fig. 9e:** ControllerActivity.kt

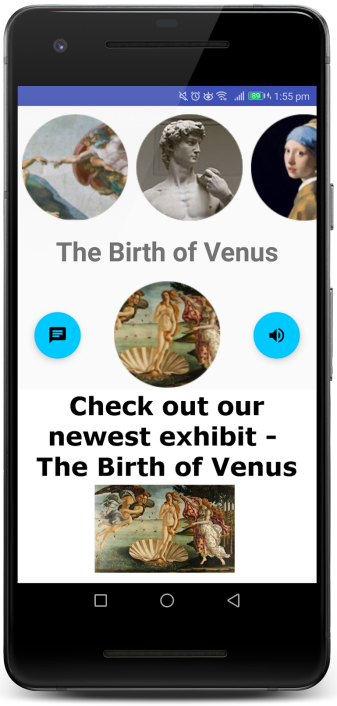



**Fig. 9f:** FollowerActivity.kt

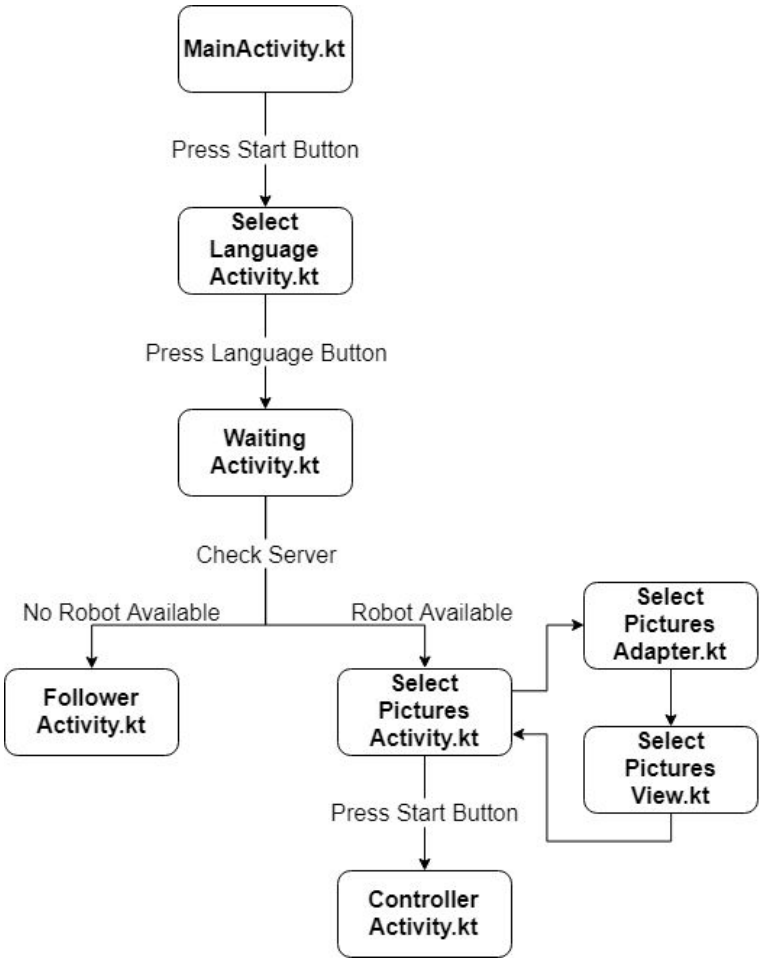



**Fig. 10**: UML Activity Diagram of Kotlin Classes for the App

The code on the app is divided into the classes pictured in Fig. 10. Upon opening the app, MainActivity (Fig. 9a) is started. When the user presses the "Start" button, they will be taken to the SelectLanguageActivity (Fig. 9b).

Once the user has selected their language, they will be taken to WaitingActivity (Fig. 9c), this activity will check the server to see if a robot is available. If one is available, they will be allowed to select pictures in the SelectPicturesActivity (Fig. 9d). Once the user has selected the pictures, they wished to go to and pressed the "Start Tour" button they will be taken to ControllerActivity (Fig. 9e).

If no robots are available, the user will be allowed to follow an existing tour in the FollowerActivity (Fig. 9f).

Most classes are self-contained as their users interfaces are relatively static. However, the SelectPicturesActivity requires an adapter and custom view as the UI needs to update when the user selects an item from the list and when the user queries the array of paintings.

The User Interface was made using Anko (Anko, 2018). We chose to develop the UI using Anko because it is faster, requires less code, and is more readable than using traditional XML layouts.

### 2.4.3 Single Controller Mode

Single controller mode allows for a controller (Fig. 9e) and a potentially unlimited number of followers (Fig. 9f). A controller chooses the paintings the robot will go to and can cancel and skip paintings, send the robot to the toilet and exit, change the robot's speed, and stop the robot. Followers can follow the tour although they do not influence the robot or the paintings the robot will be stopping at, they are free to leave the tour and join the tour as they wish, and they can also see the descriptions and etas for upcoming paintings. We allowed for unlimited followers as on peak days in museums; there may not be enough robots available, hence to better utilise resources, users can follow an existing tour (in their language).

### 2.4.4 Dual Controller mode

Dual Controller mode works the same as Single Controller mode, but there are two controllers. We allowed for this option so that two people could select the paintings they wish to on their device, and then the robot would go to the union of the paintings they selected.

### 2.4.5 Speech-to-Text

Speech-To-Text is used to allow the user to communicate with the app by searching for paintings and asking for recommendations in the SelectPaintingsActivity.kt. Speech-to-Text API (Google Cloud Speech API, 2018) was used to parse what the user says and obtain the top 10 things that the API thinks was said. The app then pattern matches to see if any key word matches any of the words the API thinks was said. Key words are any of the unique words in the paintings, the artist or some translation of “new”, “best”, “recommend”, or “popular”. If there’s a match, then the painting or a subset of the paintings is displayed in Fig. 9d, otherwise the user will be notified using a toast and text-to-speech.

### 2.4.6 Text-to-Speech

Google’s Text-To-Speech API (Cloud Text-to-Speech API, 2018) was used to read out the descriptions of art pieces when the user arrives at the artwork or on demand. The purpose of this is to allow for users who may have difficulty reading the ability to hear about the art pieces. We decided to use the Google text-to-speech API as we found that it worked very reliably on Android devices, and it allows for the dialect to be changed programmatically, so the speech sounds natural no matter what language the user selected. Another reason is due to the popularity of Google’s Text-To-Speech, users are more likely to be familiar with the voice and find it easier to understand.

## 2.5 System Structure

Fig. 11 gives an overview of how the app, server, and robot communicate with each other, and their software components.

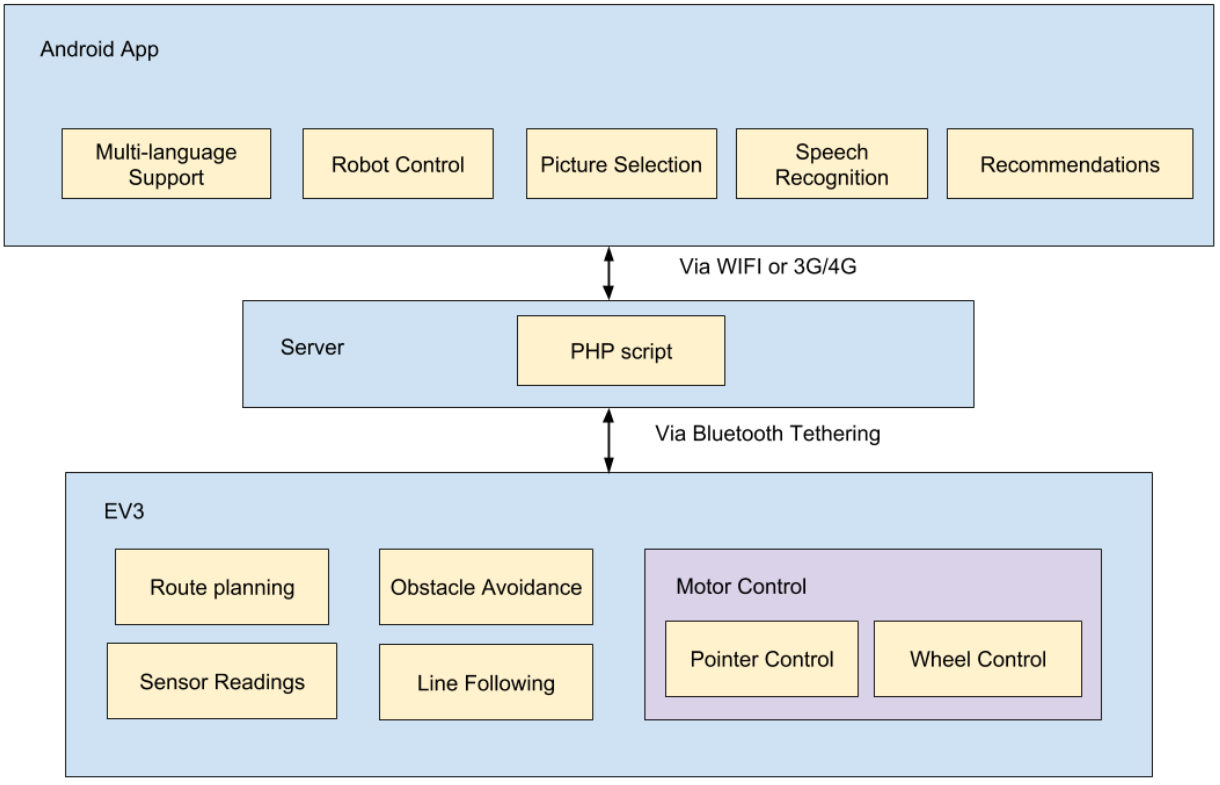



**Fig. 11:** System Structure

# 4 Software Testing

We have considered a thorough quantitative testing plan using both functional and non-functional testing, and the results were recorded. The tests combined a balanced set of tests carried out by guest users and team members.

## 4.1 Integration

To ensure integration went smoothly, before development we established our basic functional requirements of each component to ensure that we all are aware of what needed to be created and the specification. When it came to integration at least one person who contribute to each subsection was present, ensuring that it went smoothly. Once integration was finished, testing was done to ensure no errors were created during the integration process and that the tests that previously passed still pass. If new features were introduced - such as notifying the user on the android device when an obstacle is encountered - we ensured that regression testing using existing tests took place to ensure that we did not introduce bugs or negative side effects.

## 4.2 Software Reliability Testing

We have done software reliability testing of the individual components of the robot (Fig. 12b and Fig. 12c) and the Android app (Fig. 12a and Fig. 12d). These tests were done to ensure that the individual components met the requirement.

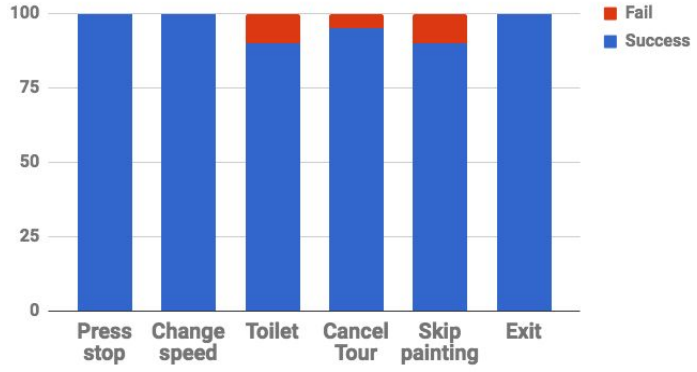



**Fig. 12a:** Software Reliability on the User Interface

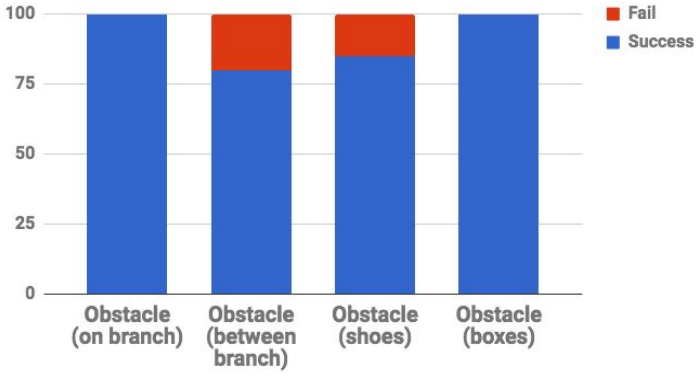



**Fig. 12b:** Software Reliability on Obstacle Avoidance

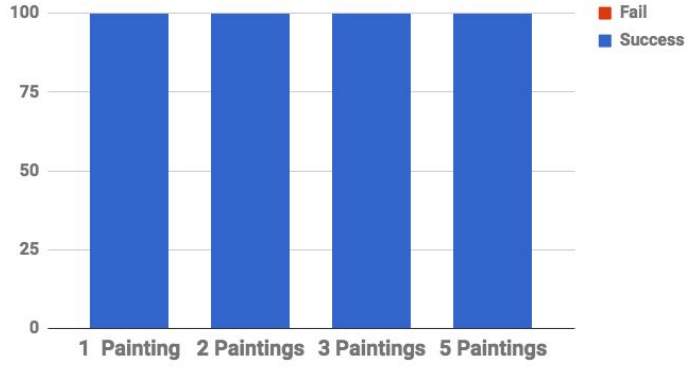



**Fig. 12c:** Software Reliability on Route Planning

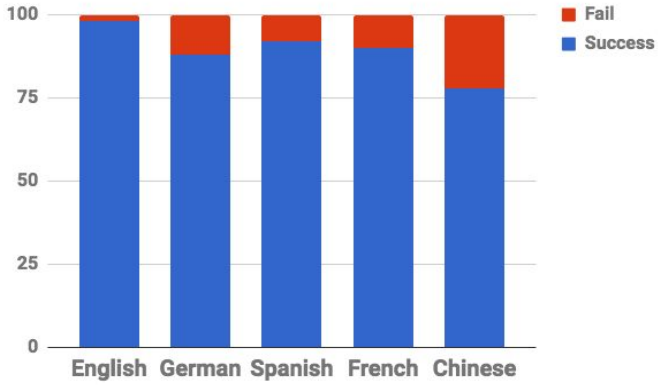



**Fig. 12d:** Software Reliability on Speech-to-Text

## 4.3 Integration Testing

After testing the individual components, we combined them and had 20 trial runs of the entire robot system with different paintings selected (Fig. 13a) and the server communication with the app and robot (Fig. 13b). This is to ensure the individual components worked as a whole.

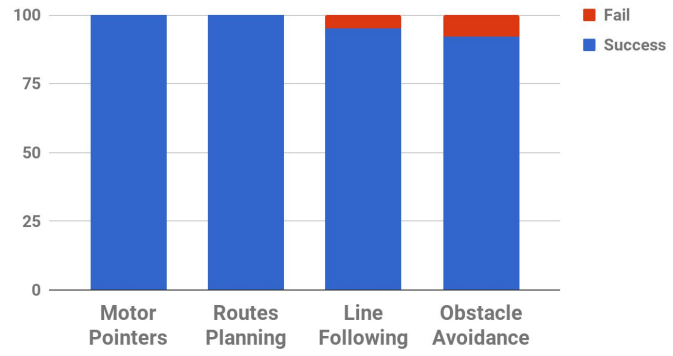



**Fig. 13a:** Integration Test for the Robot

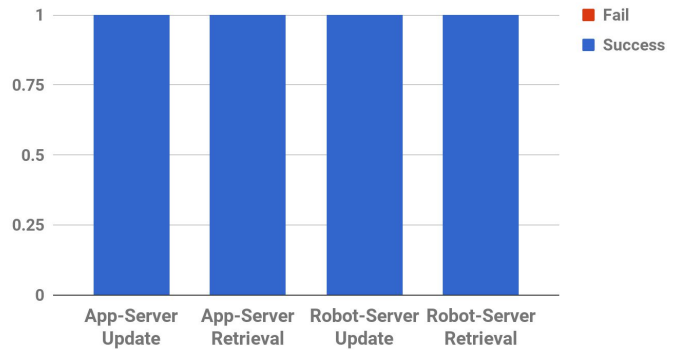



**Fig. 13b:** Robot-to-Server and App-to-Server testing

## 4.4 Full System Testing

After testing individual components and integration tests, we ran tests on the whole system from the start of a user story with different modes (Fig. 14).

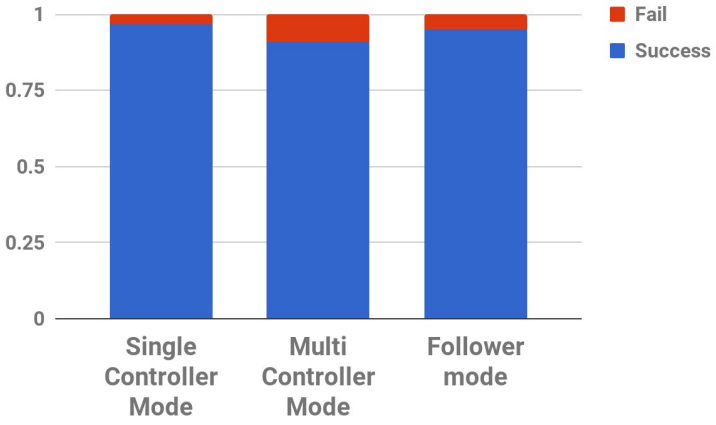



**Fig. 14:** Full System Testing

## 4.5 Usability Testing

We conducted usability tests with the app by giving users the app and seeing if it is intuitive without any explanation. We took this feedback to reevaluate the user interface to be more intuitive as per the feedback from the sample users. Some examples of user feedback that was implemented is asking the user if they are sure when exiting the app, so they do not exit accidentally and text-to-speech notifications when they arrive at the exit and toilet.

## 4.6 Compatibility testing (Android versions and phone versions and tablets)

The android app has been developed to run on Android SDK version 17 or above. We tested on different brands of Android phones and tablets to ensure that the app is displayed correctly and functionality is maintained.

# 5 Reflections

## 5.1 Noteworthy Strengths and Weaknesses of the Final System

**Strengths**:

- **Effective navigation** - efficient route-planning and line-following using the custom line sensor contributed to smooth and relatively quick navigation around the mock museum. Once working, the robot seemed to perform very reliably and in a way useful to users.
- **Intuitive interface** - throughout the project, we had been taking feedback from other people and constantly improving the user experience. The resulting application has an effective interface that provides good access to our system's features.
- **Custom line sensor** - this feature differentiates our team from many another line following robots. It increases the maximum speed possible with Lego sensors by at least 150% and has much smoother line following.

**Weaknesses:**

- **Suboptimal communication protocol** - in the final implementation the updates were relayed by constantly polling the server, which was quite taxing on the infrastructure and could even trigger safety mechanisms on the available hosting solutions (see section 5.2).
- **Limited obstacle avoidance** - the obstacle avoidance system worked well, but only in a narrow range of possible problems. There are some situations that the system cannot cope with - e.g. obstacle avoiding between two obstacles. However, solving these would have been very challenging with the resources at our disposal.
- **Inflexible navigation** - the implemented solution proved to work very satisfactorily. However, it requires physical modification of the working environment and as such could not be as easily deployed in different locations.

## 5.2 If We Could Start Over

None of the members of the group knew how to use PHP or had any server experience, and so when we made the PHP script that ran on the server (see section 3.3), it would simply send and display a string of POST requests on a webpage. This meant that both the EV3 and the App would need to poll the server about 60 times a minute for the user to have a smooth experience. None of us realised that it would cause quite a strain on the server. We realised this too late, and we didn't have enough time to redo the server and the communication code on the EV3 and Android side, hence, we just had to make do as the code was working. This wasn't so bad when we only had one app, but as soon as we added multiple phones, the problem became more apparent. The constant polling was also quite straining on the app where we had threads constantly running in the background. If we were to do the project again, we would have learnt how to do a server the way it was intended by utilising a messaging protocol such as MQTT. This would've meant a more reliable web server, less logic in the background of the app making it easier to debug, and possibly a better usage of our time.

## 5.3 Performance On Final Demos

### 5.3.1 Final Client Demo

On the day the robot worked well. Although we had two minor oversights with the app - in follower mode the speech was overlapping, and the text was missing from the current picture. These minor issues were fixed straight after the client demo, and the application was tested more thoroughly. Hence they did not reoccur in the investor demo and the app performed as expected.

### 5.3.2 Investor Demo

On the day of the final demo, the robot incorrectly detected a branch on a curve. An investigation indicated that the branch detection algorithm was tuned to a different environment (with higher background-to-line contrast) and so gave a false positive in this case. Other than this, we felt the demo went smoothly, and the robot worked as intended.

The investors seemed to think RoboTour was a viable solution to an acknowledged problem; we received a lot of positive feedback regarding the idea of the robot. It seemed like the visitors could relate to the problems in accessing the museums' resources fully and appreciated our approach to solving it.

We won the technical innovation runner-up prize - Sponsored by Robotical. The judges seemed to be very impressed with the smoothness of the custom line sensor and how it improved the robot.

### 5.3.3 Visitors at the Stand

During the trade fair, we had a significant amount of traffic at our stand. This was partially due to our robot navigating around a miniature track all day after our demo. The purpose of this mini demo area on our stand was to allow the judges who did not have an opportunity to view our pitch to see the robot in action. We found that a robot navigating on top of the table attracted a lot of interest from the general public and fellow peers. We received a lot of positive feedback about the robot, custom sensor and Android app and most seemed to have experienced the problems in museums which RoboTour has been designed to solve.

## 6 References

GitHub. (2018). *Kotlin/Anko*. [online] Available at: https://github.com/Kotlin/anko [Accessed 9 Apr. 2018].

The Verge. (2018). *99.6 percent of new smartphones run Android or iOS*. [online] Available at: https://www.theverge.com/2017/2/16/14634656/android-ios-market-share-blackberry-2016 [Accessed 9 Apr. 2018]

Tim Eckel. (2017). *Arduino New Ping Library*. [online] Available at: https://bitbucket.org/teckel12/arduino-new-ping/overview [Accessed 9 Apr. 2018].

Microchip. (2017). *Atmel AVR GNU Toolchain 3.5.4*. [online] Available at: http://distribute.atmel.no/tools/opensource/Atmel-AVR-GNU-Toolchain/3.5.4/ [Accessed 9 Apr. 2018].

*Google Cloud Speech API*. (2018). *Cloud Speech API Documentation | Google Cloud Speech API | Google Cloud*. [online] Available at: https://cloud.google.com/speech/docs/ [Accessed 9 Apr. 2018].

*Cloud Text-to-Speech API*. (2018). *Cloud Text-to-Speech API Basics | Cloud Text-to-Speech API | Google Cloud*. [online] Available at: https://cloud.google.com/text-to-speech/docs/basics [Accessed 9 Apr. 2018].

Gilles-bertrand.com. (2018). *Dijkstra algorithm: How to implement it with Python (solved with all explanations)? | Gilles' Blog*. [online] Available at: http://www.gilles-bertrand.com/2014/03/dijkstra-algorithm-python-example-source-code-shortest-path.html [Accessed 11 Apr. 2018].

Skiena, S. (1990). Dijkstra's algorithm. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley*, 225-227.