

**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 of: 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 System Structure Overview

Fig. 1 shows how the app, server, and EV3 communicate with each other, the software components they contain and how each layer interacts with each other. The Android app is used by the user(s) to communicate with the robot, the EV3 is the computer on the robot which controls its action and the server is used to mediate instructions between the EV3 and Android app.

|  |
| --- |
|  |
| **Fig 1:** System Structure |

# 

# 3 Components

## 3.1 Robot Hardware Components

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Fig. 2:** Labelled view of RoboTour | **Table 1:** Robot Components   |  |  | | --- | --- | | **Quantity** | **Item** | | 1 | LEGO Ultrasonic sensor | | 2 | LEGO colour sensors | | 2 | HC-SR04 Ultrasonic sensors | | 1 | Custom Line sensor | | 1 | Motorised Pointer | | 2 | Drive wheels | | 1 | Arduino Sensor hub | | 1 | Lego EV3 | |

The robot (See Fig. 2) 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 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.

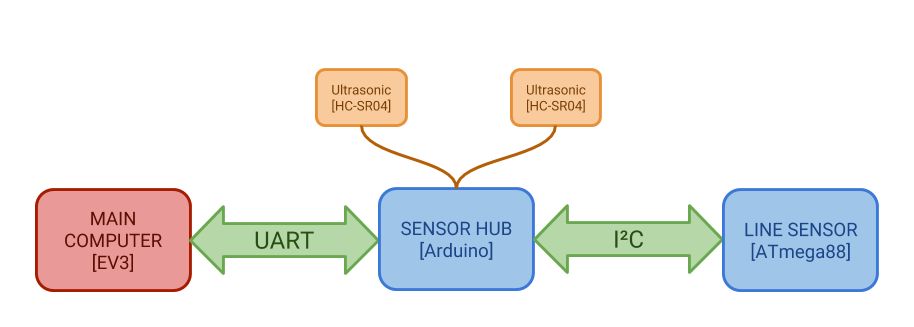
The differential drive platform was chosen due to the mechanical simplicity and intuitive control. Using two motors means that the robot can exercise control over only two degrees of freedom out of three on the plane. The trade-off was the lack of ability of translational movements in any direction besides the one the robot 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 all other elements so that it is clearly 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 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 3).



**Fig. 3**: Sensor expansion schematic

## 3.2 Robot Software Components

## 3.2.1 Programming Language

The main computing node of the robot is the LEGO EV3 running an ev3dev system, which is based on Linux Debian Jessie.

The system provides an easy and robust access to the hardware with a number of high-level functions. There is a number of 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 (e.g. 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/or develop analogous solutions for the different platform.

### 3.2.3 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 (approximately £300 each due to the equipment required) could be too high for less popular galleries. Even if the price is not an issue, the setup could be very difficult as some buildings are listed and modifying them to implement such 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 understand the reason for our chosen algorithm, figures 4a and 4b compare PID controller with a boolean line follower. The workings of PID The following paragraphs explain each of the three parts of the algorithm.  The proportional part is responsible for turning the robot at 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 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. |  |  |
| **Fig. 4a**: Without PID | **Fig. 4b**: With PID |

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

### 3.2.4 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.  The 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. 5:** Map of the Museum Environment |

### 3.2.5 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. 5) is imported as a graph (using python dictionary) with vertices representing branches or paintings and with edges representing the action required to move to the neighbouring vertex. This allows the graph implementation method enables Dijkstra 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 graph search algorithms were considered such as A\* search algorithm, and Viterbi algorithm. Although A\* algorithm and Viterbi algorithm are faster than Dijkstra algorithm when there is a large amount of vertices, they require more complex implementation and in our prototype there are very small amount of vertices thus the speed difference is negligible so we have chosen Dijkstra algorithm. Furthermore, we found existing solutions for Dijkstra algorithm and we tailored it to suit our graph implementation of the map.

The Dijkstra algorithm is run with the robot’s position and the selected stack of paintings. The closest painting in the stack and its path to reach to the painting are returned to be executed by the robot. Then, this closest painting is removed from the stack so that the Dijkstra algorithm can be run with the position of the closest painting (robot’s new virtual position) and the stack to find the second closest painting. This repeats until the stack is empty. Then the order of the paintings is uploaded to the server so that the user interface in the Android app updates the carousel order.

### 3.2.6 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.

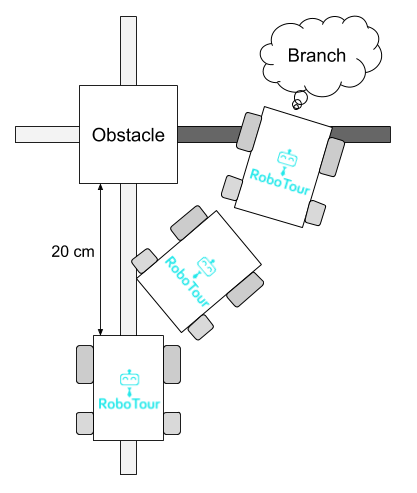
### 3.2.7 Wall following mode

Considering the obstacle may have various sharp edges, one of the best ideas for the robot to avoid obstacle is to keep a constant distance from the obstacle and move along the sharp edge. This is shown in Fig. 6.

When an obstacle is detected, the robot will turn 90 degrees in a pre-programmed direction. The two HC-SR04 ultrasonic sensor at the side are used to detect the distance to the obstacle. The robot will keep the **Fig. 6**: Obstacle Avoidance

distance at 11 centimeter, 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.

### 3.2.8 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. 7). 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.

**Fig. 7:** Black Line detected

### 3.2.9 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” the robot will return the pointer to the set position and start navigating to the next painting.

## 3.3 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 connman 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 connman WiFi driver. We decided reliability was more important for our use case hence we chose Bluetooth tethering.

## 3.4 App

According to The Verge, 81.7% of smartphones in the market 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 time restrictions, we should focus on developing a prototype for only one platform.

## 3.4.0 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). 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 ensurest the app has approximately 20% less code than that would be required with Java, making the app easier to debug and adapt.

## 3.4.1 User Interface

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
| **Fig. A:** MainActivity.kt | **Fig. B:** Select  Language  Activity.kt | **Fig. C:** WaitingActivity.kt | **Fig. D:** SelectPictures  Activity.kt | **Fig. E:** Controller  Activity.kt | **Fig. F:**  Follower  Activity.kt |

|  |  |
| --- | --- |
| **Fig. 8**: UML Activity Diagram of Kotlin Classes for the App | The code on the app is divided into the classes pictured in Fig. 8. Upon opening the app MainActivity (Fig. A) is started. When the user presses the “Start” button, they will be taken to the SelectLanguageActivity (Fig. B).  Once the user has selected their language, they will be taken to WaitingActivity (Fig. C), 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. D). Once the user has selected the pictures they wish to go to, and pressed “Start Tour” button they will be taken to ControllerActivity (Fig. F).  If no robots are available, the user will be allowed to follow an existing tour in the FollowerActivity (Fig. E).  Most classes are self contained as their UI are fairly static, however the SelectPicturesActivity requires an adapter and custom view as the UI needs to update when the user selects an item on 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. |

### 3.4.2 Multi Controller Mode

Multi controller mode allows for up to two controllers and an unlimited number of followers. A controller can cancel and skip paintings (upon mutual agreement with the other controller), send the robot to the toilet and exit, change the robot’s speed and stop the robot. Followers are able to follow the tour although they have no influence on 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, they can also see the descriptions and ETA’s 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 own language).

### 3.4.3 Single Controller Mode

Single user mode works the same as multi-user mode but there is only one controller. We allowed for this option so if museum visitors prefer to make all selections using one device they can.

### 3.4.4 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 write 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. D, otherwise the user will be notified using a toast and text-to-speech.

### 3.4.5 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 join in during the speech. We decided to use the Google text-to-speech API as we found that it worked very reliably on Android devices, 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.

# 4[.0](#_17dp8vu) 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.

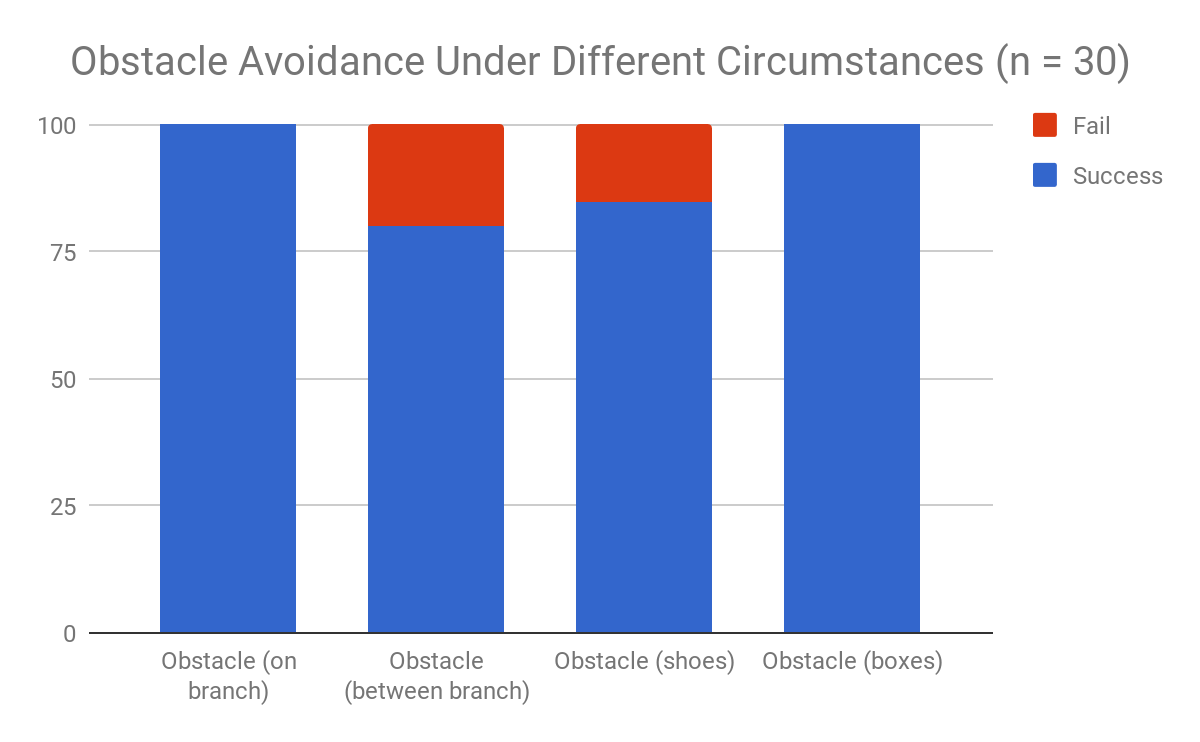
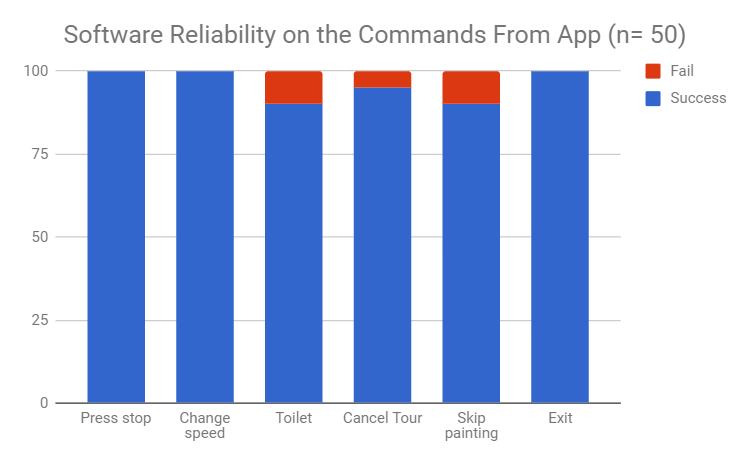
## 4.1 Integration

To ensure integration went smoothly, before development we created documents summarising the basic functional requirement of each components 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 was doing each subsection was present to aid in integration, 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.

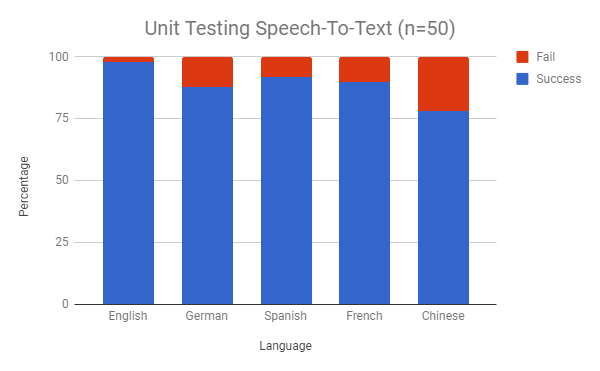
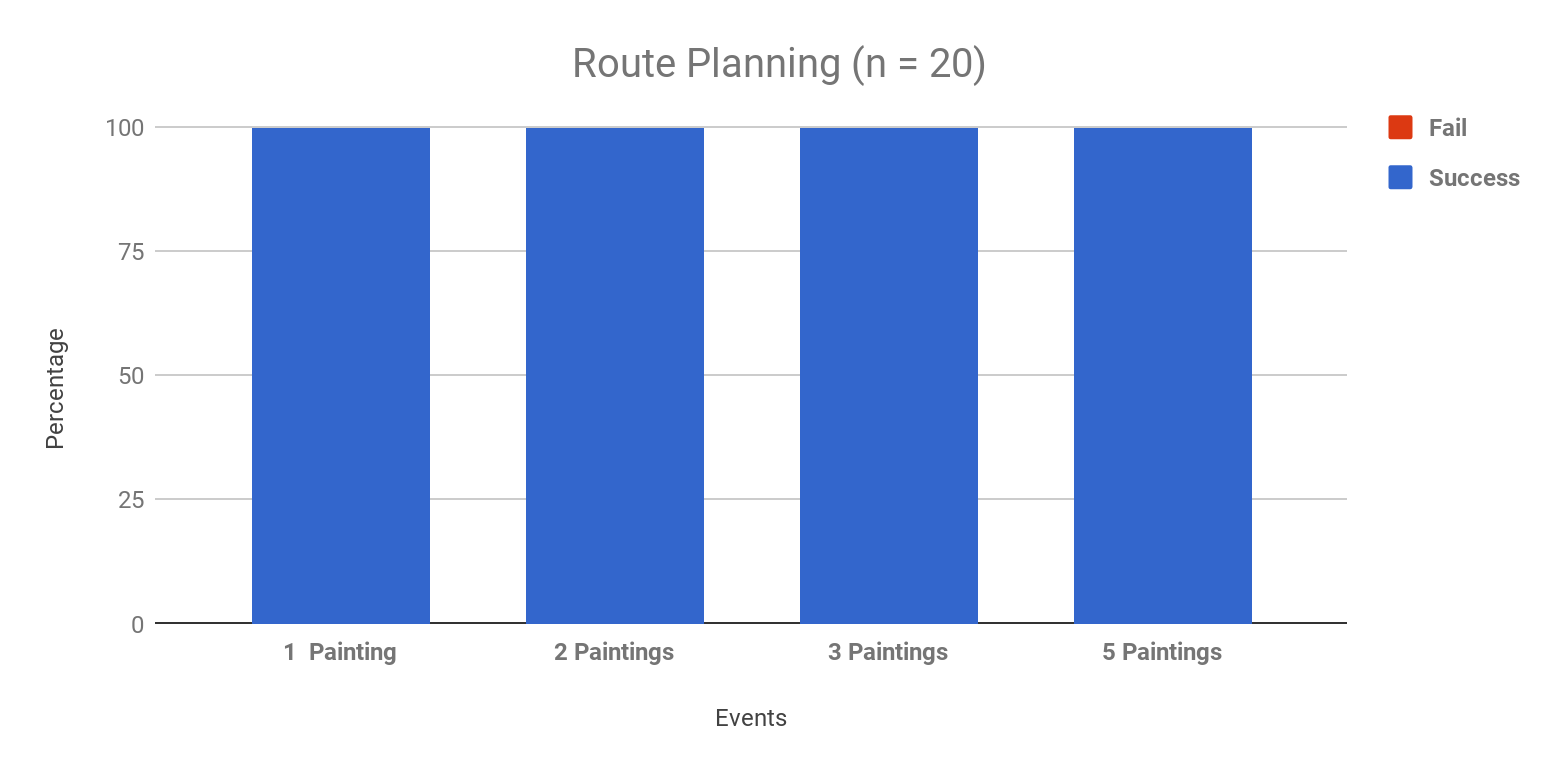
### 

### 4.2 Software Reliability Testing

We have done software reliability testing of the individual components on the robot (Fig. 9b and Fig. 9c) and the Android app (Fig. 9a and Fig. 9d). These tests were done to ensure that the individual components met the requirement.



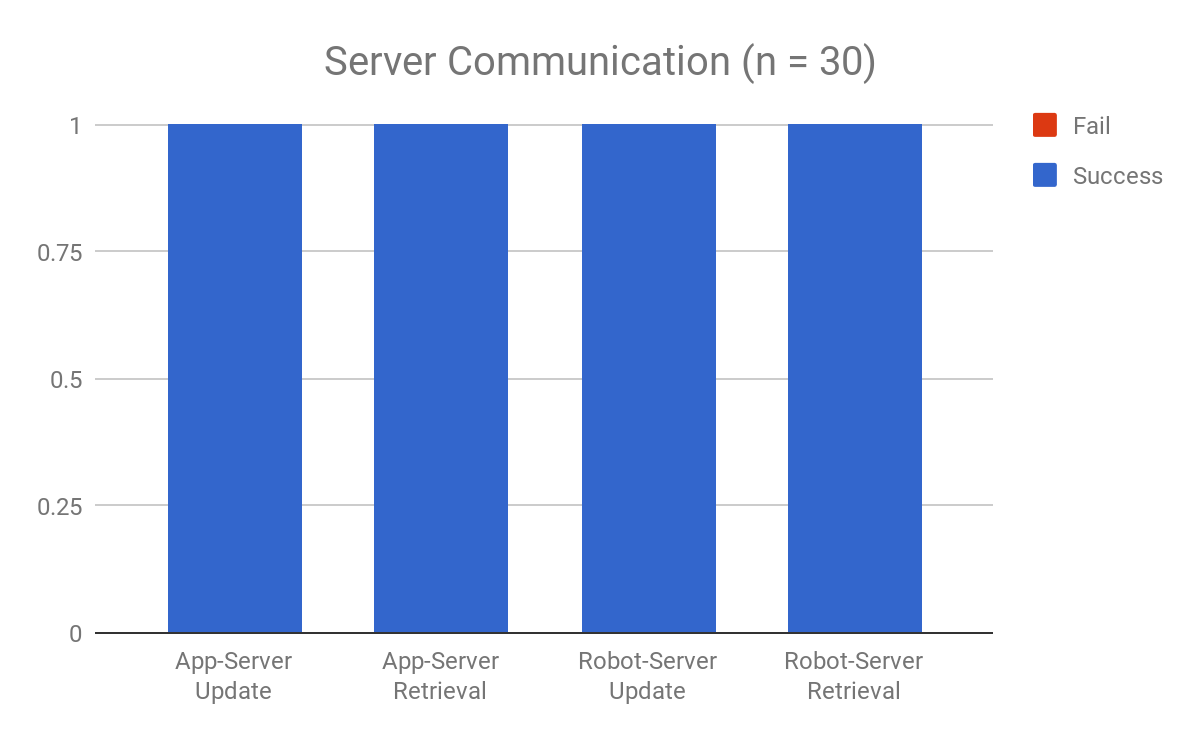
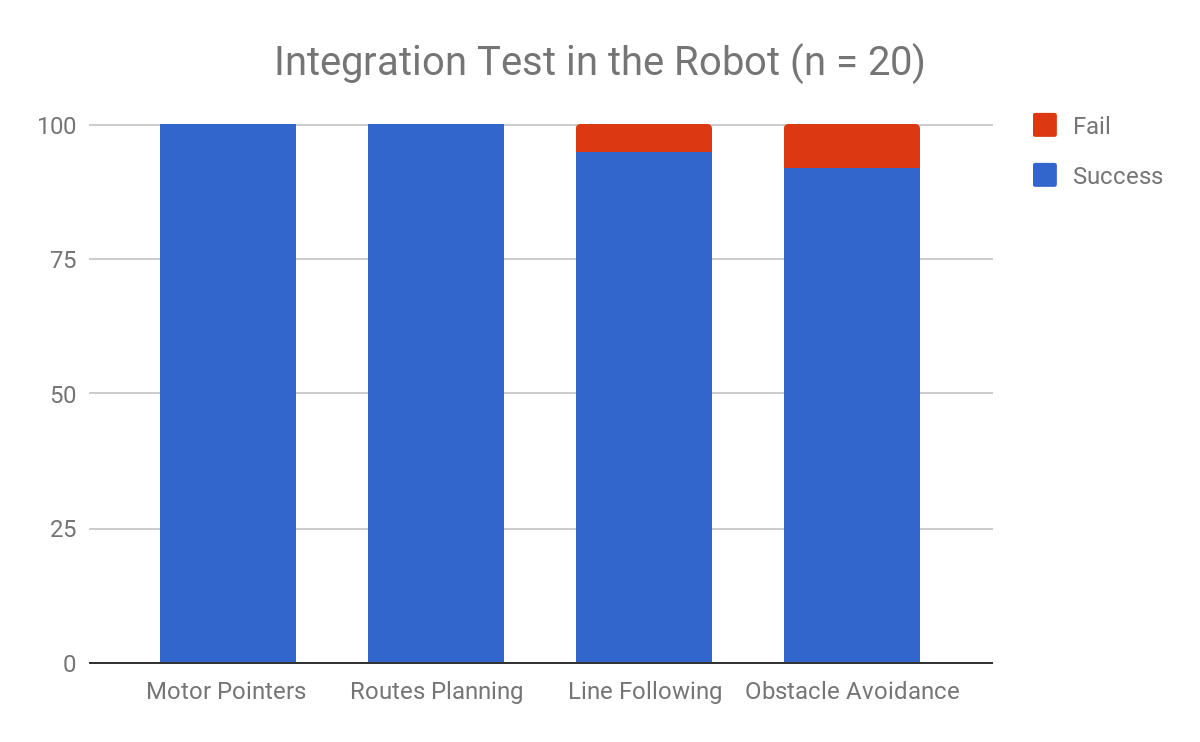
**Fig. 9a:** Software reliability on the user interface. **Fig. 9b:** Software reliability on obstacle avoidance.



**Fig. 9c:** Software Reliability on route planning **Fig. 9d:** Software Reliability on speech to text

### 4.3 Integration Testing

After unit testing the components, we combined the components together and had 20 trial runs of the entire robot system with different paintings selected (Fig. 10a) and the server communication with the app and robot (Fig 10b). This is to ensure the individual components worked as a whole.



**Fig. 10a:** Integration test for the robot **Fig. 10b:** Robot to server and app to server testing

### Chart

### 4.4 Full System Testing

After unit testing individual components and integration tests, we ran tests on the whole system from the start of a user story with different mode (Fig. 11).

### 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 **Fig. 11:** Full system with different modes

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. When in certain activities we were passing intents to the operating system requesting the screen remain on regardless of user settings to allow the app to notify the user upon arrival at paintings, we discovered that some Android devices - such as Huawei devices - disregard this request as they have modified the stock Android features.

# 5.0 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 a 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 makes up for an effective interface that provides good access to our system’s features.
* **Custom line sensor** - this feature differentiates our team from many other line following robots. It increases the maximum speed possible with Lego sensors by at least 150% and 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.
* **Limited obstacle avoidance** - the obstacle avoidance system worked well, but only in a narrow range of possible problems. There is a number of situations that the system cannot cope with - e.g. Obstacle avoiding between two obstacles - however solving them would be very challenging without restructuring the system.
* **Inflexible navigation** - the implemented solution proved to work very satisfactory. 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 so 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 required in the background of the app making it easier to debug, and possibly overall better usage of 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. 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 at solving it.

## 5.4 Achievements

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.

We also added a few bonus features not outlined in the project plan such as a custom line sensor, multi user support and follower mode. These features were added to enhance the robots usability and appeal to museums.

# 6.0 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].

*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].

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