University of Edinburgh
School of Informatics

System Design Project
Technical Report

Group 5 (Electric Sheep):

Abbi Culemann, Viktorija Lukosiute, Stelios Milisavljevic,
Wojciech Nawrocki, Claire Purslow, Alex Roy, Sorin Vladu

April 18, 2018

Contents

1 Introduction

1.1 Objective

Project *Hermes* is our attempt at changing how delivery robots work. Instead of just delivering objects from point A to point B, our system is an integrated, autonomous inventory management solution. It is indeed capable of moving things around, but the movement itself is automated and optimised for minimal energy consumption. The task of sorting items and remembering where they are is offloaded from people to the server and robot network. It can be used in a factory or workshop environment.

Constructing such a system is straightforward in some aspects, and challenging in others. Our ideas about the project and its design evolved to account for constraints in time and resources.

1.2 Design evolution

Different designs we considered for *Hermes* can be seen on *Fig. 1.* From the start we knew that it is not viable to build a full-scale version of our robot capable of transporting items in a real-world environment. We decided to instead construct a scaled-down prototype with all the functionality, but smaller dimensions.

The initial idea was to build a portable storage robot that either picks items up above itself (a) or one which stores items within its body (b).

We quickly came to the conclusion that design a does not meet our requirement of being able to pick items up from multiple levels in a single shelving unit, while design b would be complicated to implement due to requiring empty space in the robot internals. Moreover, the full-scale version would need to be voluminous enough to store useful items inside.

We settled on a version of design c, where the robot has a moving forklift in front, but one with different dimensions and an altered body. The robot works by moving into a multi-level shelf with the forklift in the right position, and then reversing out of it with an item on the lift.

Following week 2, there were no dramatic changes to the concept. The initial design of the robot was iterated on and improved throughout the course. Relevant design decisions are outlined in this document.

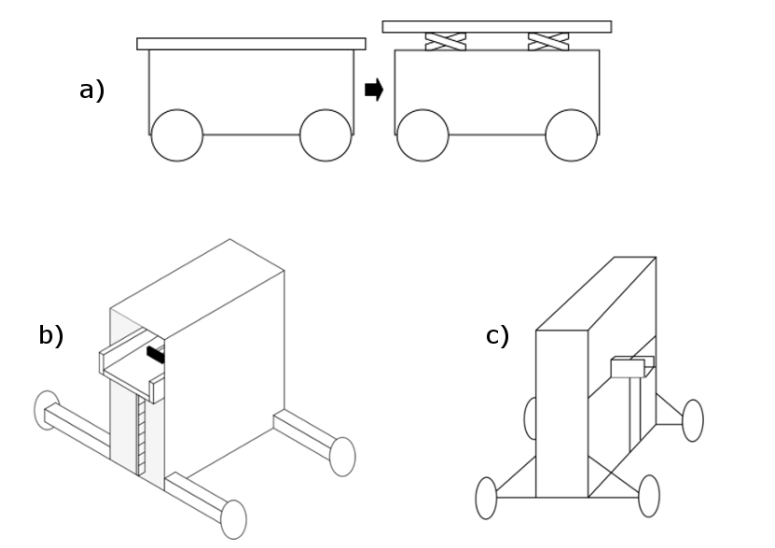
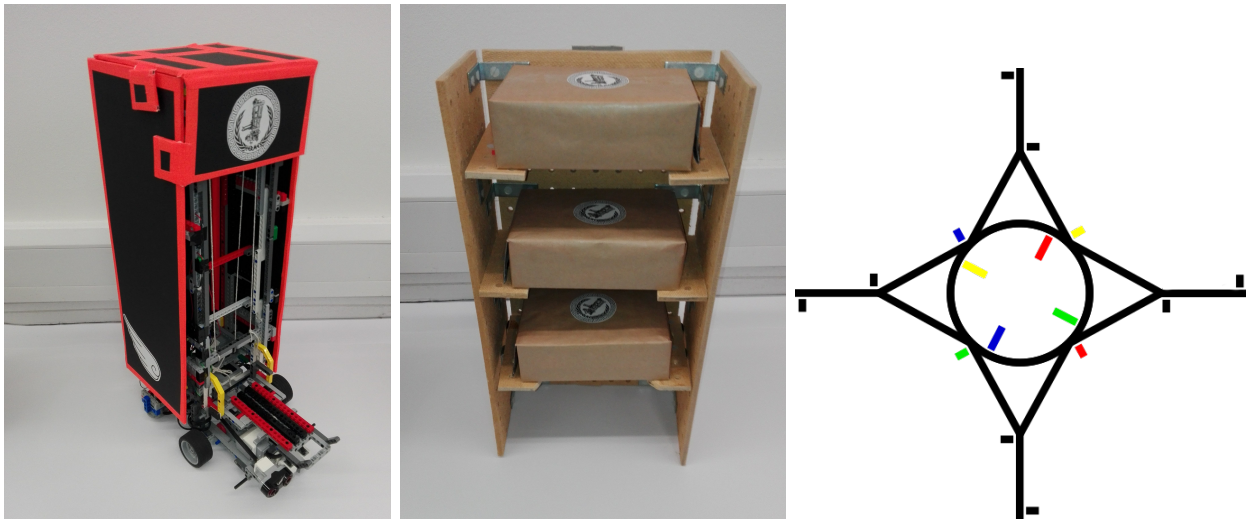


Figure 1: Evolution of robot design

1.3 Final product

After 3 months, Hermes became a fully functional robot with a working back-end server to manage item allocation and storage and a front-end client for users to interact with the system. The robot is able to pick up and deliver items to their respective locations while navigating over a line network on the ground in a small-scale testing environment. *Fig. 2* shows the final version of our product together with a junction schematic.



(a) Robot (b) Shelves (c) Junction

Figure 2: Final product

2 Design

2.1 Shelving

The first robot design had shelves paired to be of same height as the robot and the containers to be placed on top of the shelf stands. Since everything would have to be in a horizontal fashion, such a system would take up too much space and was discarded (example in Figure 3). As the lifting mechanism evolved, so did the shelving units.

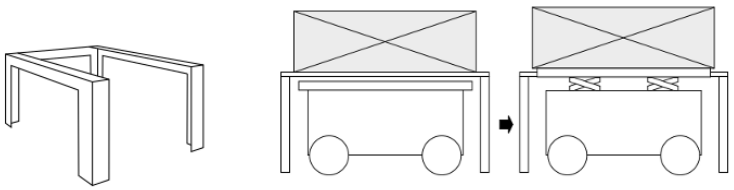


Figure 3: Sketch of horizontal shelving unit

Soon it was clear that there needed to be two separate parts to the shelving system: the internal shelves which only the robot has access to, and the drop-off point which was accessible by both humans and robots. The shelving units at the current scale could feasibly hold up to three shelves.

The drop-off point is designed to the same specification as the lowest shelf in the shelving unit. This maintains the lowest possible number of sensors needed on *Hermes* to be able to navigate lifting and dropping subroutines.

Both the drop-off point and shelving units are made with peg board, two- or three-hole right angle brackets, plastic nuts and screws.

In the shelving unit there are two side boards of the same dimensions, a back board of the same height but a larger width, and each of the three shelves have two small pieces of peg board for the containers to rest on. Two two-hole brackets are used for each piece of the shelf rest, while four three-hole brackets are used between each side board and back board (Figure 4).

Figure 4: Shelves

The drop-off point has the same structure, however it only has one set of shelf rests. The side and back boards are also shorter.

In a full scale system it would be possible to add more shelves to the shelving units. It would also be possible to enlarge the boxes to be able to carry much larger tools and materials.

The containers are cuboids, with one end that can be opened, and can be fastened by Velcro. At the early stages of the project the boxes were held together with tape, however Velcro is closer to clasps that would be used in the deployed product. The boxes are made of firm card, as the weight of the boxes is minimised. If the boxes need to be sturdier, plastic or wood could be used, maybe even carbon fibre or metal. All of these would require a slightly different access mechanism to the boxes since the materials are not as easily folded as card.

2.2 Server

2.2.1 Front-end

The front-end is a web app hosted by the server. This is the primary way in which users interact with the system. The front-end is separate from the back-end and communicates with the back-end through the JSON API (see Figure 5). This separation was chosen to allow for code to be developed and maintained independently by different developers. Also, development of the back-end in this way allows expansion to different front-end clients (such as a mobile app).

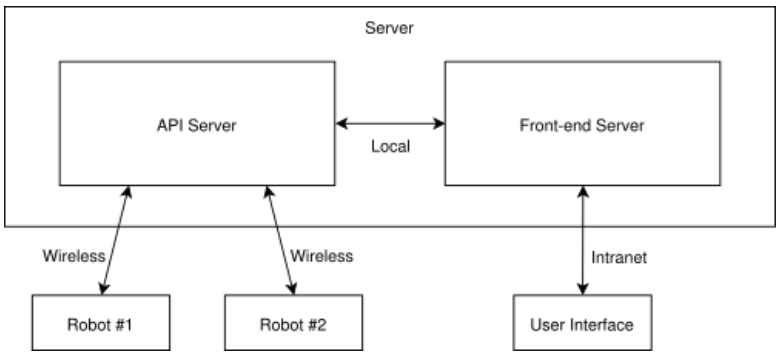


Figure 5: High Level Software Overview

The user interface runs in the user's web browser with interactivity facilitated by JavaScript and AJAX generated using the React JavaScript library. React was chosen due to the way it employs reusable "components" to build up UIs. This allowed us to a maintain consistent "look-and-feel". The team

started with very little web development experience. React's high level of abstraction and shallow learning curve allowed us to produce a clean front-end in the given time.

The user interface of the web app is shown in Figure 6.

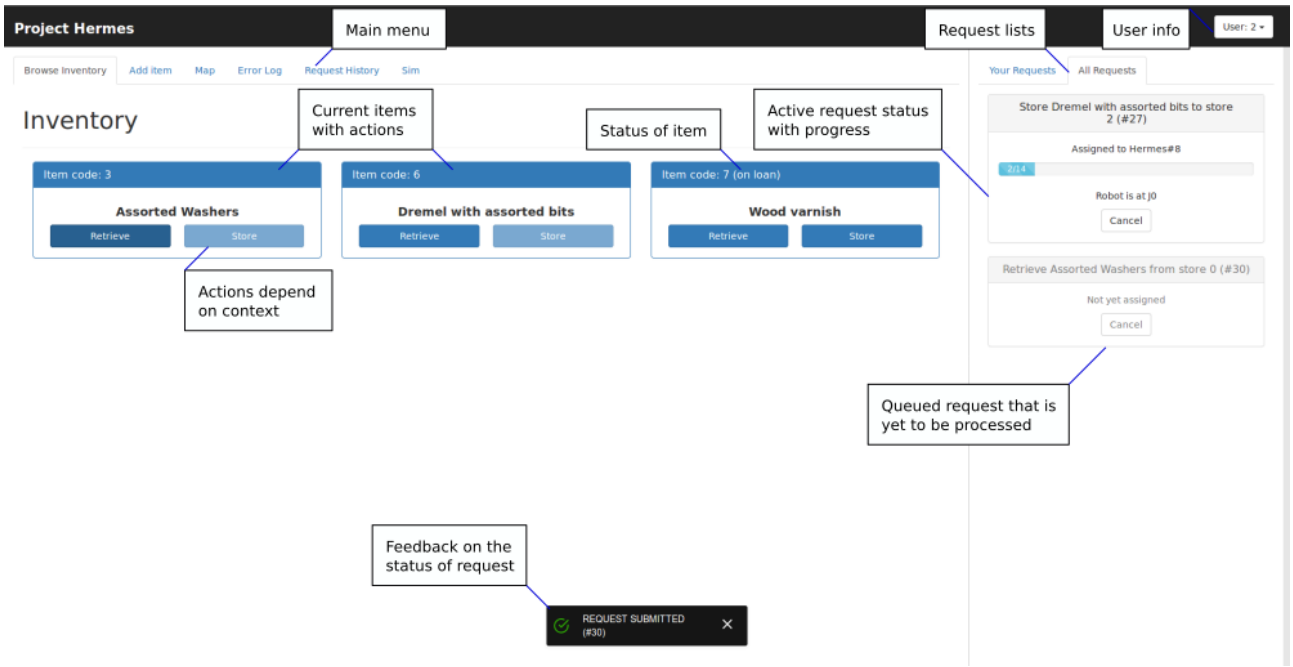


Figure 6: User Interface with Descriptions

2.2.2 Back-end

The back-end server is the back-bone of this system. It handles the logic behind the inventory keeping, manages all the requests made by the users, and orchestrates the robots. Node.js was chosen to be used for the server because it allows operation of multiple concurrent connections without incurring the cost of thread context switching making it easier to handle multiple robot connections simultaneously.

The connection between the robot and the back-end is made through websockets as the robots and the server are connected to the same local network. An active connection (as opposed to a polling-based system) provides more robust interaction with the robot because it's easier to detect unexpected problems like a robot disconnection. The messages exchanged were initially plaintext commands that were harder to extract information from, thus they were replaced by JSON formatted objects so that the information passed would be easier to manipulate and check against a defined schema. The relation between the robot and the back-end can be regarded as Master/slave model of communication. The back-end sends the request to be completed by the robot and then receives the result from the robot.

Communication between the front-end and the back-ende is made possible using RESTful API as mentioned in the front-end section. Subsequently, the Express.js web application framework was used to building the API. The API is separated into different categories. Each is handled by a different Expres.js router. This helped modularise the code and lower the coupling of the system making it easier to maintain. During the development, the API was fully documented (see Appendix A) so that the developers working on the front-end could ensure compatibility with the back-end. The API was evolving regularly, adding new interfaces and changing existing ones after feedback from the front-end developers. To keep track of changes and allow independent development of the front-end from the back-end, the *Postman* API development environment was used. This allowed the back-end to simulate HTTP requests from the front-end and the front-end to simulate responses from the back-end whilst keeping the API synchronised between the two development teams.

The management of multiple robots and the inventory keeping are all handled by the back-end. User

requests are received through the API and stored in an "active requests" queue where they are waiting to be processed by a robot. The active requests are executed by either an idle robot or a robot that has just finished processing another request (and so is about to become idle). When a request is being processed, it is added to the "processing requests" list. This list gets repeatedly updated with the position of the robot and the progress of the request (see section 2.3). At the same time, the back-end informs the robot whether the request it is processing has been cancelled.

Initially the system was made so that an item that is currently being used by a user (and thus not in the inventory), could not be requested by another user. However, this felt inefficient for real use as the requestor would have to keep checking if the item is returned. The solution was to have the request wait in the "active requests" queue until the item was to be stored by the current occupier and then the instruction would change from *retrieve item from inventory*, to *transfer item from workstation to workstation*.

In a system crucial for the smooth operation of a factory, it is important to keep records of the all requests and errors. The logging is made for accountability purposes and troubleshooting. It is important for an admin to have as much information as possible. Additionally, the back-end provides special interfaces for administrators. For example, an admin is able to insert an item directly in the inventory, unlike regular workers that have to request an empty box from the inventory when adding a new item.

2.3 Robot client

The client running on the robot can be considered as a slave to the back-end server. It was designed this way so that the client can be kept as simple as possible and that each robot does not need to know the global state of the system.

The client is written using Python 3 as it has the best documented API for ev3dev and there is no need for compilation on an ARM processor, which is the case for C++.

When the robot initially connects to the server, the map is fetched and saved. The map has all the information regarding nodes, junctions and optimal exits for each route. This is used for calculating the path for any given request. At transitions between lines, junctions, and endpoints the robot knows its current location. When the robot receives a new request, the path is calculated by traversing the graph from the current location to the destination location and each time the robot is at a junction it takes the optimal exit until the robot is at the destination.

A feature that we deemed necessary and that would set us apart from similar robots was the ability to cancel a request once the robot has started processing it. Initially, we intended to have the instructions execute in a thread and when the cancellation occurred, the thread would stop. This solution, however, proved to be insufficient as the robot would struggle to know exactly at which point the instruction was cancelled making it difficult to plan its next actions. Ultimately, our solution decomposes a higher

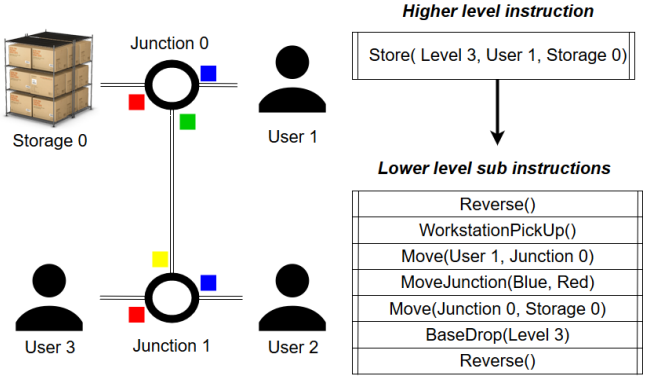


Figure 7: Higher Level Instruction Decomposition

level instruction sent from the server, into smaller independent sub-instructions (see Figure 7). Each of these instructions would then be added into a queue. Then the queue would be processed in order and each sub-instruction executed would have their opposite instruction (to reverse the action) added to a "cancellation" stack. When finishing each sub-instruction, the robot sends its location and progress to the server and checks if the request was cancelled. If a cancellation was made, processing of the queue is halted and instead the "cancellation" stack is executed, returning the system to its previous state. This design decision proved helpful as it made the implementation of the movement code straight forward. The movement code had to only be implemented for each sub-instruction instead of a complicated higher-level instruction. This significantly simplifies the logic needed by abstracting the whole decomposition and execution process. Furthermore, when something unexpected occurs with the robot, like losing the line and not finding it again, the system handles the problem by throwing a specially tailored exception from the failed sub-instruction. The failure and its cause is then sent to and logged by the server and shown to the users.

While implementing the client, a separate sub-instruction class was used. It did not use the movement instructions that needed the EV3 to run, but instead printed diagnostic messages that mimicked the movement of the robot (e.g. "Moving from Node 3 to Junction 3"). Thus, the robot's client could run on a personal computer, accelerating the design, implementation and testing times. This version of the robot's client proved to be useful as it was also used for the simulation of multiple robots (see section 3.3).

Finally, it should be noted that the client was implemented before the server so that the movement sub-instructions could start being developed and tested as soon as possible. For this reason, the groups that were working on the sub-instructions were provided with a simplistic version of the back-end (until the server was developed enough to be integrated) that sent hard-coded requests directly to the robot without the need of a front-end. And so the development of the sub-instructions was independent of the work done for the server.

2.4 Robot hardware

2.4.1 EV3 controller

All logic processing on the robot is done on the EV3 "brick" controller placed close to the base of the robot. The controller contains an ARM [1] CPU and runs a fully-featured Linux distribution provided by the `ev3dev` [2] project. All custom logic specific to this project is written in Python [3] , allowing for a quick edit-test-debug prototyping cycle. The controller contains several ports for connecting peripherals used in subsystems throughout the robot:

- two colour/reflectance (toggleable mode) sensors (input)
- an ultrasonic sensor (input)
- an Arduino microcontroller (input-output)
- two large motors (output)
- a medium motor (output)

Important hardware components are highlighted in *Fig. 8.*

[1] `https://en.wikipedia.org/wiki/ARM_architecture`
[2] `http://www.ev3dev.org/`
[3] `https://www.python.org/`

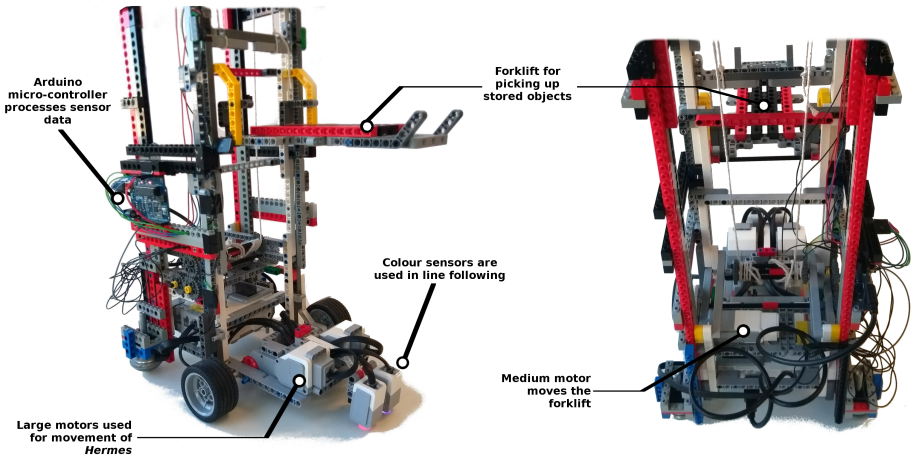


Figure 8: Outline of important hardware components

2.4.2 Movement system

Individual *Hermes* robots navigate around their environment using a network of lines. Early in the project, a number of navigation systems were explored including use of an "eye-in-the-sky" and other computer vision based approaches. However, we opted to build a structured, simpler environment for the robot to ensure reliable operation and minimise complexity. For a factory setting, we felt that a line-following system was reasonable in terms of the amount of required additional infrastructure. We also felt a robust system was achievable with the resources we had available.

The line network is made up of 2 types of component: "lines" and "junctions". These are arranged to form a spanning tree over the factory with routes between each end-point (i.e. workstations and shelves). Instead of lines meeting at single points we designed junctions that resemble roundabouts. This junction design was chosen to enable higher speed transfer of items in the factory (maximise time in motion) and to allow multiple robots to navigate the network simultaneously. Specifications for the physical format of the line network can be found in Appendix B.

The robot uses 2 EV3 colour sensors to navigate the line network. At any one time, the robot uses one to follow the line using a PID algorithm and one to detect coloured markers adjacent to the lines. The markers are used to inform the robot when it is approaching junction entrances, junction exits, and end-points.

In our initial system design we had decided to use caterpillar tracks for the movement of the robot. Initial testing showed us that the turning part of the movement system was rather inaccurate but the base was sturdy and solid which was our initial goal in designing the robot base in order to sustain the the fact that we would build a robot quite large in height. After we implemented our first attempt at the PID we realised that the caterpillar tracks errors in their turning would affect our movement and hence we decided to do more testing on what would be the best option as a replacement for the tracks. Our initial replacement were 4 large wheels which we thought would help out in the accuracy of the turning rate of Hermes, however we have found an even better solution with just two regular wheels in the front and two casters in the back of the robot. The reason why we chose to use casters was because our robot would never have to go in reverse for lengthy distances and hence we could trade the speed and accuracy of moving in reverse for the precision of forward movement and turning.

After we implemented the initial PID controller we have indeed found errors in the line following system. Since it was an early stage we couldn't determine whether those were external errors or the algorithm failing because of the bad tuning. Hence we invested more time into manually tuning the algorithm until testing showed a sufficient result. The mathematical function describing our PID algorithm is the following:

$$u(t) = K_p e(t) + K_i \int_0^t e(t')\,\mathrm{d}t' + K_d \frac{\mathrm{d}e(t)}{\mathrm{d}t}.$$

2.4.3 Lift and grabbing mechanism

The delivery functionality of the robot requires it to have a mechanism capable of picking up boxes and placing them onto shelves - the lifting mechanism. Its hardware consists of:

- a tall scaffold
- a lift capable of moving vertically along the scaffolding, with a magnet attached to the side
- a motor attached to the scaffolding
- a set of gears and a string converting motor motion to lift movement
- magnetic Reed switches placed at precisely defined positions along the scaffolding
- an Arduino microcontroller

The Arduino and motor are connected to the main EV3 controller.

The Arduino microcontroller fulfils a simple function of measuring digitised (0 or 1) conductivity through all of the Reed switches and reporting their status back to the EV3 controller. The Reed switches pass signal through when there is a magnetic field nearby, suppressing signal otherwise (Figure 9). The Arduino turns these readings into a 32-bit mask. The mask is sent via USB to the EV3, which then unpacks the sensor values by reversing the masking process. This allows the EV3 to see the status of all switches - specifically, which of them have the magnet, and hence the lift, nearby.

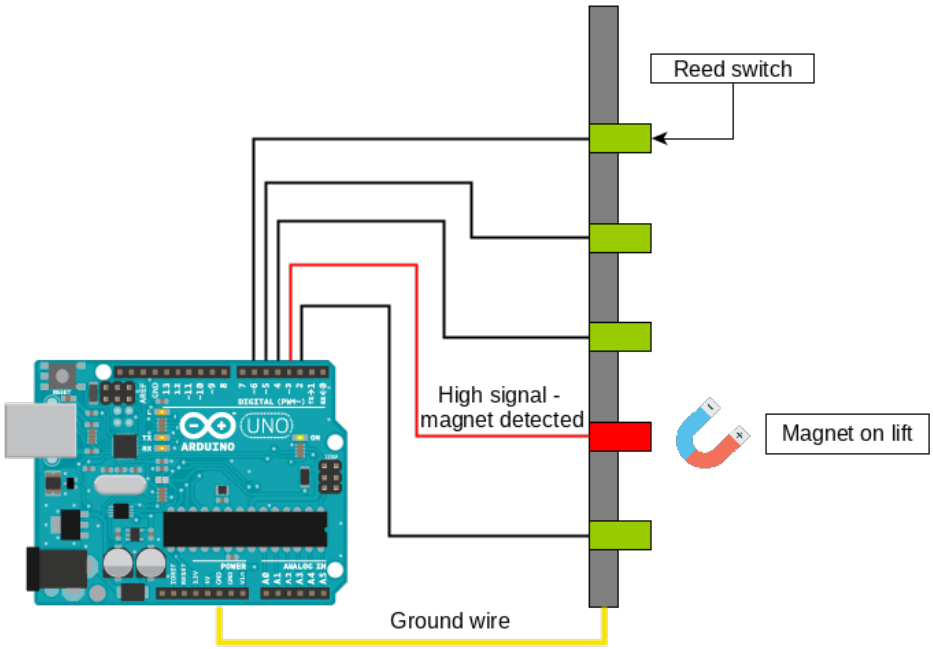


Figure 9: Schematic of magnetic sensing mechanism

The bulk of logic processing is done on the EV3 controller. When a lift positioning instruction is issued, the EV3 instructs the motor to begin moving the lift upwards or downwards in the direction of the desired position. While it is moving, Reed switch signal is polled continuously from the Arduino. When a specific pattern in the signal is detected, the lift stops. We noticed this pattern and the fact that it indicates that the lift is necessarily in the right position during quantitative testing of the sensors (see Section 3.1.1).

One thing that should be noted is that we assume the only magnet nearby is that on the lift. The lift movement mechanism could be intentionally or accidentally tricked into thinking the lift is somewhere

else through the use of external magnets. In a factory environment, this would require shielding of the mechanism from external fields.

3 Testing

3.1 Quantitative testing

To understand the impact of signals from sensors and outputs of peripherals on the robot's control functionality, we employed extensive quantitative testing. This is the most detailed type of testing and focuses on specific parts of the robot.

3.1.1 Lifting mechanism sensors

Figure 10 shows the behaviour of signal from Reed switches installed along the lift scaffolding (see Section 2.4.3) when the lift position (height) is varied. After creating the graph, we noticed a clear pattern - every sensor sees two peaks in signal when the lift is nearby. The valley between the peaks corresponds to the lift being exactly at the sensor position. We utilised these results and encoded them in the lift movement logic to guarantee robust and precise positioning.

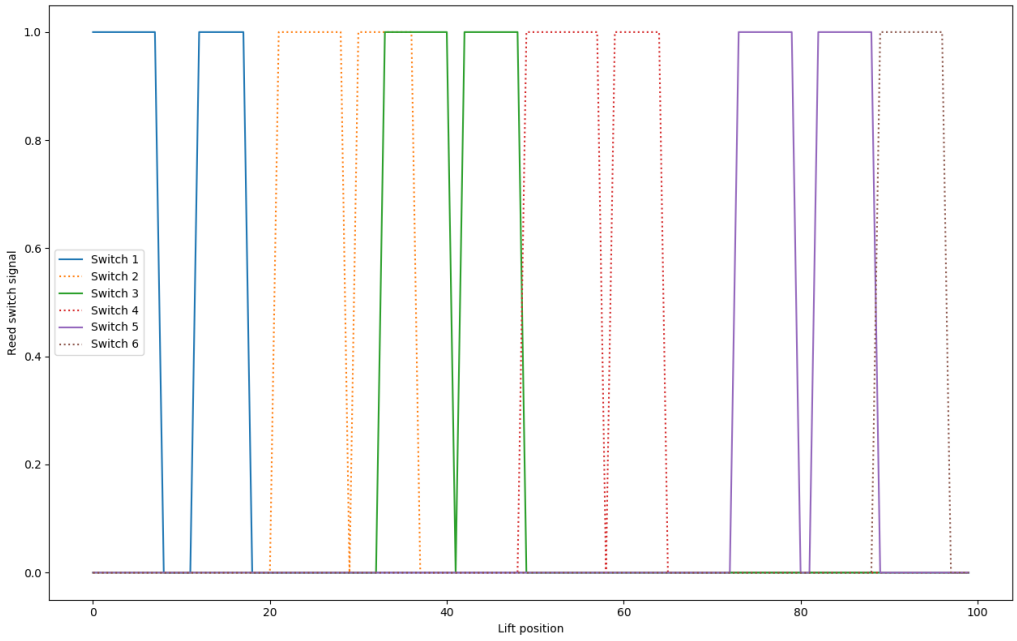
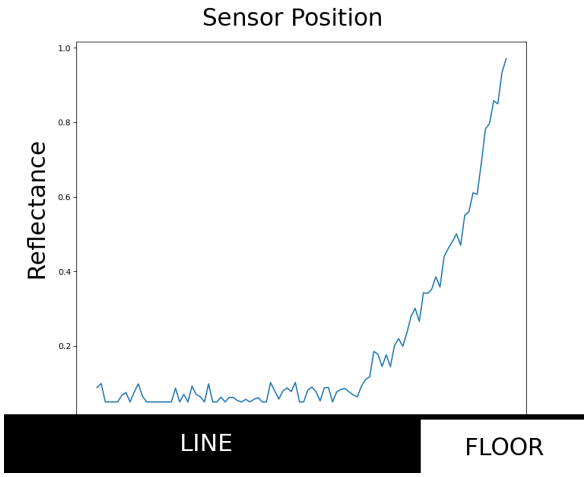


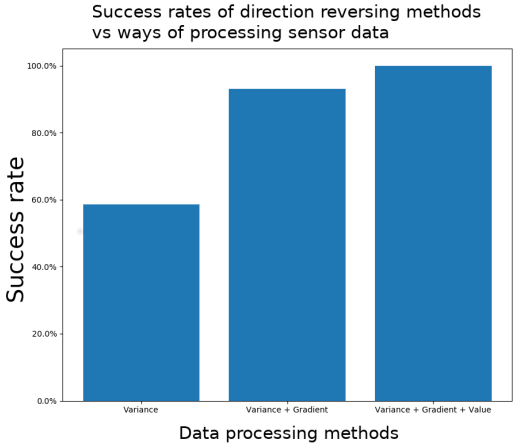
Figure 10: Signals from Reed switches vs lift position

3.1.2 Ground colour sensors

The main signal utilised in ground movement control comes from two colour/reflectance sensors placed at the front of the robot (see Section 2.4.2). To understand how to best process these signals to achieve precision in line-following, we graphed the measured reflectance vs. the position of the sensor (*Figure 11a*). The measurement was made by repeatedly moving a sensor rightwards, from a black line onto a white background, and then averaging the results.



(a) Floor reflectance under light sensor vs horizontal sensor position



(b) Success rates of reversing function vs ways of processing sensor data

Figure 11: Results of light/colour sensor testing

From the graph we made several hypotheses on how the data can be processed to achieve robust movement, all based on storing the last few measurements in a circular buffer and then computing statistical measures on its contents.

To illustrate an example of our methodology, figure 11b displays the increase in success rate of the `Reverse` [4] instruction as we incorporated more ways of processing sensor data.

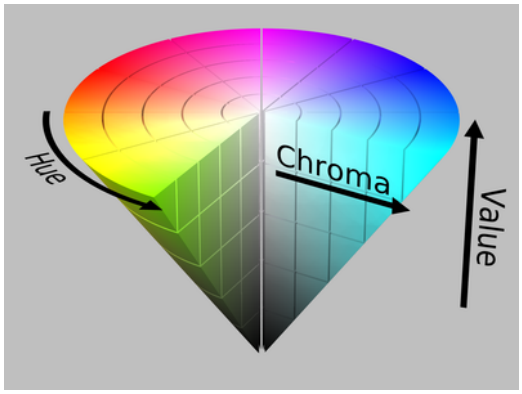


Figure 12: HSV colour space

First, computing the variance can be utilised to detect line-floor edges. Reversing by only taking variance into account in the logic has a success rate of about 60%, because a big enough random fluctuation can trigger the variance threshold. Next, we extended the logic to also consider the gradient of the last few values in the buffer. A high gradient indicates passage from a dark to a light background, and a low one indicates the opposite. Processing data in this way improved the success rate to about 90%. Finally, checking the reflectance value under the sensor after an executed rotation brought the success rate close to 100%.

Another aspect of ground movement that we tested quantitatively was the detection of colours using the same light sensor in colour mode. About halfway through the project, a lot of spurious failures started to occur (e.g. reporting blue when the sensor was over black). Initially we tried doing the colour detection ourselves by transforming raw RGB sensor readings into HSV space (Figure 12) and looking at the Hue component, but due to poor results we resorted to trying another sensor. That did help, indicating that our initial sensor was either calibrated incorrectly or had a manufacturing error.

3.2 Integration testing

To verify that all of the robot subsystems and external software (the server) work together, we performed integration testing. As part of our plan, we described several use cases for the robot. The integration testing process then followed the procedure below:

[4] `Reverse` turns the robot around on a line by 180°

1. Setup environment for a selected use case.
2. Run the robot through the use case.
3. As soon as a failure occurs, triage it, then use quantitative testing on the failing component to fix the issue.
4. If the use case has been tested enough, select next use case and go to step 1. Otherwise go to step 2.

To test specific behaviours, we selected use cases which would stress them the most. For example, to test line-following functionality we would pick use cases with the longest and most complicated paths over the line network, as well as ones involving rare edge cases, such as optimizing junction traversal by skipping movement over its inside if it's unnecessary.

3.3 Simulation

The robot client code (section 2.3) was designed such that it could simulate robots on the same computer as the server. This allowed development and testing of the robot-server communication without access to the hardware of the robot. In particular, from early in the project, we were able to simulate multiple robots concurrently communicating with the server.

Initially, the simulated robots differed from the real ones only in lacking calls to hardware-specific `ev3dev` code. Later, this was expanded to actually simulate the robots navigating the map. This was achieved without changing the fact that each robot is aware only of its local surroundings, but is unaware of the state of the entire system.

The multi-robot scenario requires only a two additional behaviours to prevent collisions between robots:

1. When two robots are heading in opposite directions on a collision course, one pulls to the side to allow the other to pass.
2. Robots must not enter a junction when another robot is directly blocking their path.

The sensing needed to achieve these behaviours is simulated on a separate part of the server back-end that runs in parallel to the existing back-end. It employs an additional websocket connection only used by simulated robots. Before simulating a "step" of motion, the robot queries the server which checks whether this motion is compatible with the motions of other robots and responds accordingly.

The simulation can then be visualised on a map on the front-end by creating an additional Expres.js router to expose the simulation information (see 2.2).

4 Evaluation

4.1 Strengths and weaknesses of the system

Strengths:

- fully autonomous - requires no human interaction besides deliver/store requests
- simulation testing has shown that it can scale up to a large number of robots

Weaknesses:

- external factors play a large role in success rates - robustness needs improvement

4.2 Final Day Performance

For the final day presentation, we selected a three of small demonstrations to showcase different functionality of the system.

Firstly, we showed a user in the factory requesting an empty box to store an item into the system. The robot connected to the server and the font-end displayed the correct information. It then correctly proceeded to pick a box off the shelf, navigate the line network and deposit the box at the user's workstation.

Secondly, we showed how a different user can request the item that is now displayed in the inventory (on the UI). Their request was queued by the robot and there was wait time whilst the the first user had the item. We then showed the first user returning the item and how the system instructed the robot to transfer it straight to the second user. During the delivery of item, the second user cancelled their request and the robot recognised it. The item then began to be returned to storage instead. During the return journey, however, the robot lost track of the line and began to veer off course. The system recognised this failure, stopped the robot, logged it, and informed the user. Whilst we resorted to a video to present the correct operation, we showed how the error was handled and displayed the log in the front-end UI.

Thirdly, we showed a simulated, multi-robot system running on the same server infrastructure with an expanded line network. Due to using a development build, an error of visualising the robot positions was displayed at the start of simulation. This was an unfortunate symptom of a bug in a library we used in the development build. After this, the simulation could be seen running correctly.

4.3 Lessons Learned

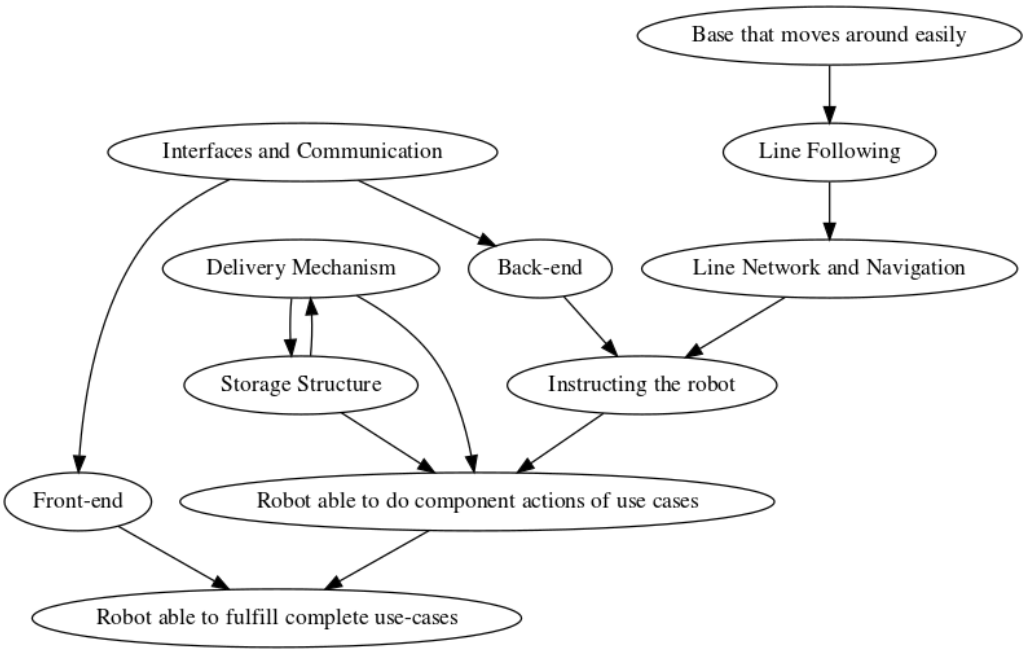


Figure 13: Overall Project Structure

4.3.1 Team Structure and Organisation

We began the project by identifying very general "stages" of the project and how they depended on each other (see Figure 13). This allowed us to make 3 "sub-teams" within the team that could work simultaneously. The lines and base team, lifting mechanism team, and interface and communication team. These were more an effort to categorise tasks than team members with individuals being on

multiple teams and contributing to whatever tasks they could. Over the course of the project, the categories converged as they were integrated into the final product.

Throughout the project, we opted for a "flat" team structure with no rigid hierarchy. On the whole, this worked well for us with little frustration. Although, there were no formal chains of responsibility, team members who had done significant work on particular aspects of the system would "organically" be responsible for that aspect. However, we all stayed flexible – occupying different roles as needed at different times. As a team we aimed to be respectful and patient with each other.

Organisationally, we did run into a few difficulties. As a team, we repeatedly underestimated the amount of time required to complete tasks. This was usually due to either inexperience or under-specification of what the task actually required. This also often lead to an uneven workload where we would work longer hours nearer deadlines.

On reflection, this perhaps could have been avoided. Whilst we would set deadlines for individual tasks, we would rarely schedule exact times at which the completion of the task would be carried out. We could have scheduled work in such a way that we would get revised time estimations (after starting) early enough that we could plan around tasks that require more work than expected.

4.3.2 Technical Challenges

Reflecting on approaches to technical aspects of the project we did well to focus on maximising the simplicity of the robot. For example: we designed the line network in such a way that traversal of the network only requires combination of 2 methods, and that routes can be generated from a single Python dictionary. In general, we had a goal to achieve complex behaviour with minimal components.

If done differently, we could improve some of our approaches. For one, we did manage to create a web-based interface, but it was done through a significant and time-consuming learning curve. No one in the team had much previous knowledge in the field and it did affect the way we worked. Instead, we could have gone for an Android Mobile App since the group was more familiar with the technology. Another challenging aspect was working with the hardware because almost all of our group members were well-versed software developers. More than a few times we assumed that sensor noise and inaccuracy would not be significant enough to affect the robustness of the system and were proven wrong. Examples include spurious colour sensor readings that caused complete system failures early on and robot being unable to find lines after rotating on the spot due to detecting the grey floor as black.

5 Appendix A

5.1 API

Method	URL	Parameters	Request Example	Response Example	Description
GET	server:8080/api/requests/			[{ "action": "retrieve", "itemCode": 0, "dst": "1", "id": 1, "src": "0", "level": 1, "emptyBox": false, "title": "Retrieve sandpaper, Wet and Dry, Fine from store 0", "completed": "completed" },{ "action": "retrieve", "itemCode": 1, "dst": "1", "id": 2, "src": "0", "level": 2, "emptyBox": false, "title": "Retrieve Torque screw-driver with assorted bits from store 0", "completed": "completed" }]	Retrieve whole request history
POST	server:8080/api/requests/		{ "action": "retrieve", "itemCode": 1, "dst": "3" }		Create a retrieve request
POST	server:8080/api/requests/		{ "action": "store", "src": "3", "itemCode": 1 }		Create a store request
POST	server:8080/api/requests/		{ "action": "transfer", "src": "2", "dst": "1" }		Create a transfer request

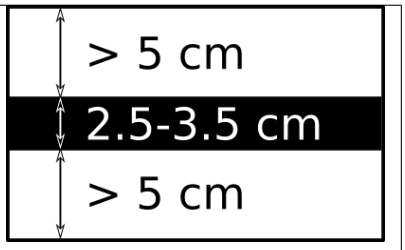
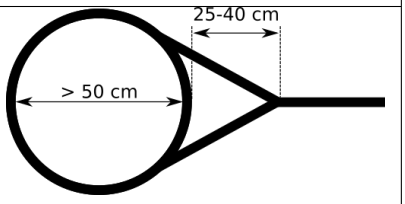
Method	URL	Params.	Request Example	Response Example	Description
GET	server:8080/api/requests/active	?user=id (optional)		[{ "action": "store", "itemCode": 1, "src": "1", "id": 4, "dst": "0", "level": 2, "title": "Store Torque screwdriver with assorted bits to store 1", "completed": "no" }]	Retrieve active requests waiting in the queue
GET	server:8080/api/requests/	?user=id (optional)		[{ "id": 3, "robotId": 1, "position": "1", "progress": { "totalSteps": 8, "currentSteps": 1 } }]	Retrieve requests being processed by a robot
GET	server:8080/api/log			[{ "id": 1, "timestamp": "2018-03 18:54:09 +01:00", "robotId": 1, "requestId": 1, "requestTitle": "Retrieve Hacksaw from store 0", "error": "Robot lost line" }, { "id": 2, "timestamp": "2018-03-30 18:54:34 +01:00", "robotId": 2, "requestId": 2, "requestTitle": "Store Hacksaw to store 1", "error": "Robot lost line" }, { "id": 3, "timestamp": "2018-03-30 18:54:47 +01:00", "robotId": 3, "requestId": 3, "requestTitle": "Retrieve Torque screwdriver with assorted bits from store 0", "error": "Disconnected while processing request" }]	Retrieve the event log

<table>
<tr><th>Method</th><th>URL</th><th>Params.</th><th>Request Example</th><th>Response Example</th><th>Description</th></tr>
<tr><td>PUT</td><td>server:8080/api/inventory</td><td></td><td><pre>{ "lastCode": 4,
   "inventory": [{
      "code": 0,
      "name": "Sandpaper, Wet and Dry, Fine",
      "location": "0",
      "level": 1,
      "inStorage": true
    }, { "code": 1,
      "name": "Torque screwdriver with assorted
      bits",
      "location": "0",
      "level": 2,
      "inStorage": true
    }, { "code": 2,
      "name": "Toothed lock washers, 30mm",
      "location": "0",
      "level": 3,
      "inStorage": true
    }, { "code": 3,
      "name": "Junior hack saw",
      "location": "0",
      "level": 4,
      "inStorage": true
    }, { "code": 4,
      "name": "Glue gun",
      "location": "0",
      "level": 5,
      "inStorage": true
    }]
}</pre></td><td></td><td>Replace the existing inventory with the given inventory (used mainly for testing)</td></tr>
<tr><td>DEL</td><td>server:8080/api/requests/{request_id}</td><td></td><td></td><td></td><td>Cancel a request</td></tr>
<tr><td>POST</td><td>server:8080/api/inventory</td><td>HEADER<table><tr><td>Key</td><td>Value</td></tr><tr><td>Req.</td><td>admin</td></tr></table></td><td><pre>{ "name": "Large Staples",
  "location": "0",
  "level": 3 }</pre></td><td></td><td>Add an item as an admin</td></tr>
</table>

Method	URL	Params.	Request Example	Response Example	Description
POST	server:8080/api/inventory	HEADER Key: Value Req.: user User: user id	`{ "name": "Small Staples" }`		Add an item as a user
GET	server:8080/api/inventory			`[{ "code": 0, "name": "Sandpaper, Wet and Dry, Fine", "location": "0", "level": 1, "inStorage": true }, { "code": 1, "name": "Torque screwdriver with assorted bits", "location": "0", "level": 2, "inStorage": true }]`	Retrieve the current inventory
GET	server:8080/api/inventory/{item_code}			`{ "name": "Electric screwdriver", "code": 1, "inStorage": true, "location": "0", "level": 1 }`	Retrieve an item from the inventory
PUT	server:8080/api/inventory/{item_code}		`{ "name": "UPDATED ITEM", "location": "0", "level": 3 }`		Update an item
DEL	server:8080/api/inventory/{item_code}				Remove an item from the inventory
GET	server:8080/api/status/idleRobotIds			[1]	Retrieve idle robot ids

6 Appendix B

6.1 Line Network Specifications

Line Specification	Lines are black with a width between 2.5cm and 3.5cm. The black lines are directly bordered by white lines with width of at least 5cm.	
Junction Specification	The junctions use lines of the same thickness as specified above. The lines must fork between 25cm and 40cm from the edge of a central circle with diameter greater than 50cm. No more than 4 lines may connect to a single junction.	
Idle Area Specification	Idle area is a line connected to a junction in the same way a normal line is. However, this line must have markers along its length as shown below in markers.	
Markers	There are black junction markers on the left and right of the line incident to a junction (travelling towards the junction).These are 0cm and 20cm from the fork, respectively.