Intelligent Robotics: Waiter Robot

Nicola Amadei, Daniel Ashe, Giorgio Iavicoli, Pavel Pricope, and Jacopo Vendramin
Team Raj
School of Computer Science
The University of Birmingham

***Abstract*—This document describes the design of a robot capable of object recognition that will act as a waiter for the people using the study spaces.**
The goal is to allow students sitting at a table to order drinks from a web interface and be served, as well as having a robot that wanders around checking for table occupancy and detecting leftover bottles to be collected by cleaning staff. The robot will pick up objects from a specified location by going there and vocally asking which object it wants to take. It will then bring the object to the specified table, and if more people are sitting there it will ask to serve them too, using vocal interaction. The robot will also return back to the served tables after a while to ask the students for a refill of their drinks.

I. Introduction

The Pioneer 3-DX is a general purpose differential-drive mobile robot platform with several options including laser and sonar sensors. It could be equipped with an onboard computer and accessories like an Orbbec depth camera. The robot is driven using the Robot Operating System (ROS) which is a framework for writing robot software. This framework provides a robust and general-purpose environment to develop robot control software, as well as universal, ready-to-use libraries like the Navigation Stack and aMCL, which help the robot navigate and localise itself within a map respectively.

The purpose of this project is to use the robot's capabilities in order to build an autonomous system that would be of use to computer science students and staff. Because of the intensive use of the tables by the Computer Science students for long periods of time, we decided to create something that will help to make the study space a more comfortable area, so we came up with the idea of the waiter robot. The robot has the ability to receive orders through a web interface while also providing information about the availability of the stock of drinks in the pickup location. The order is executed by requesting to be loaded with the product at that location, and then bringing it to the correct table, vocally interacting with the students. When he is not executing orders, the robot wanders around updating the occupancy state of the study spaces, asks previous customers whether they would like a refill, and also notifies the cleaning staff if there are any bottles left at an unoccupied table. These behaviours will be described in more detail in the following sections.

Fig. 1. Basic Particle Filter algorithm

function PARTICLEFILTER($S_{t'}, u_t, z_t$)
  $S_{t'} \leftarrow \{(x^i_{t'}, x^i_{t'}) | i = 1, ..., n\}$ ▷ Belief $Bel(x_{t'})$ is represented as a set of n weighted samples
  $S_t \leftarrow \{\}, \eta \leftarrow 0$
  for i in $1..n$ **do**
    $\hat{x} \leftarrow p(x_{t'})$ ▷ Sample from the discrete distribution given by the weights in $S_{t'}$
    $x^i_t \leftarrow p(x_t | x_{t'}, u_t)$ ▷ conditioned on $\hat{x}$
    $w^i_t \leftarrow p(z_t | x^i_t)$
    $\eta \leftarrow \eta + w^i_t$
    $S_t \leftarrow S_t \{x^i_t, w^i_t\}$
  end for
  for i in $1..n$ **do**
    $w^i_t \leftarrow \frac{w^i_t}{\eta}$
  end for
return S_t
end function

II. Related Work

Particle Filter

One of the two main components in Navigation is Localization. The ROS framework provides a good implementation of a particle filter called aMCL. It is an implementation of KLD-sampling, which adapts the size of the sample set over time [1], and does not require many parameters to tweak. The ones that influence the behaviour of a particle filter most are the minimum and maximum number of particles and those regarding the resampling step, which have particular importance in maintaining a consistent behaviour and allowing the filter to cope with the kidnapping problem [2]. The main idea behind KLD-sampling is to adjust the sample size based on the Kullback-Leibler divergence, which is a distance function from the estimated to the true distribution. The KL-distance is defined as $KL(p, q) = \Sigma^n_i p_i \cdot log2(p_i/q_i)$. This follows the intuition that too few particles cannot represent a big enough portion of the state space while too many samples can become a computational burden, especially during the filter update phase.

Motion Planning

After analysing the ROS built-in global planner, we choose to try to implement our version of it. Our first step was to try to create a ROS node able to receive a map, a

starting pose and a goal; on top of that, it runs A-Star search to find the best path. As a test, we created a Python node capable of receiving such inputs while printing the optimal path on the command line output.
We implemented two versions of the algorithm, one based on a graph and the other based on occupancy grid with the aim to compare them against each other and pick the one that performs better. The graph-based method uses nodes to represent valid space configurations while the edges define the paths between them and the cost to complete that move. Finding the best trajectory is to find the shortest path between the start and goal vertices, and the fact that only nodes representing free space are part of the graph gives it a structural guarantee that when a path is found, then it is also valid.
On the other hand, the second method consists of using an occupancy grid that divides the area into cells (pixels). These can be in one of three states: occupied by the robot, permanently taken by an obstacle (wall, chair) or free space. The best route is found by determining the shortest line that does not cross any of the occupied cells. After running a few tests on the algorithm, it was found that Python could not manage to compute paths in a reasonable amount of time (i.e. within seconds), most likely due to the high resolution of the map (4000 x 4000) and to Python's overhead in runtime performance and limitations in memory management.
Then we implemented the same algorithm in C++ and noticed that the execution time dropped to less than a second, therefore making it feasible for a real-time application.

While implementing the entire navigation stack by ourselves, we realised that we didn't have enough time to re-implement the features associated with costmaps nor to integrate all the new custom nodes that we were creating. Based on this, we choose to use the built-in navigation stack that the ROS framework offers and focus more on other custom components as well as trying to achieve a more robust result overall.

Asif et al. published a paper in which they described the design process of building a robotic waiter[3]. Their paper focused more on the electrical engineering aspect of the design, by using individual electrical components to build a robotic waiter, however we can still study and critique the interfaces that they use and the design choices that they made. Asif et al. use a component they call a "menu bar" as an interface for taking customer orders. The menu bar is made up of an LCD screen, a keypad and a Bluetooth module. Users of the service enter their order using the keypad as input, and can see their order on the LCD screen. When the customer places an order it is sent to the kitchen via the Bluetooth module, to be prepared by a human chef. This means that the robot needs to be physically close to the customer for them to place an order.

The robot navigates to tables by following lines printed on the floor of the restaurant, using 2 infrared sensors. This approach appeared to work well for the creators in the lab, however in a real restaurant it may cause problems when, for example, tables need to be rearranged to accommodate large parties and the lines on the floor can't be easily moved. The robot also has IR sensors on the left and right sides, which are used to detect tables and infer the location of the robot by counting the tables. The kitchen staff tell the robot which table to stop at by sending a table number via a WLAN network. The justification for using WLAN for kitchen-to-robot communication is that it has a higher range than Bluetooth, however the authors use Bluetooth for communication in the opposite direction which appears to be a bad design choice because they don't give a reason for using it instead of WLAN in both directions. This means the robot will need to be closer to the kitchen to send an order, but it can receive commands from the kitchen at a greater distance.

Maxwell et al. created a robot waiter that they called Alfred[4]. This robot was capable of speech interaction (both speaking and listening), identifying people (and distinguishing which of the people it identified were VIPs at the conference it was demonstrated in) and navigating busy rooms without the need to have lines printed on the floor, which was the approach adopted by Asif et al.

Alfred's speech synthesis was implemented by simply prerecording all of the things that the author's wanted Alfred to be able to say. Multiple audio files were made for each thing that they wanted Alfred to say. The alternative would have been to use a text-to-speech program, however the creators wanted Alfred to be more human-like, so they used human voice recordings instead. This was motivated by work done by Clifford Nass, in which he showed that people tend to respond psychologically to computer personalities in the same way that they respond to human personalities[5]. The idea was that if they attempted to make him more human-like, it might reinforce peoples instincts to treat him more like a human and accept him as a suitable waiter. According to the experimental analysis, this proved to be successful because people treated him as a human, referring to him by name and even trying to have conversations with him.

Alfred's voice recognition was implemented using IBM's ViaVoice SDK[6]. ViaVoice allows you to create grammar files which contain words and phrases governed by rules that define the "utterances" to be recognised. An utterance is a stream of speech that maps to a command. ViaVoice has built in features that deal with different pronunciations, making the development of the grammar file easier.

The creators of Alfred decided to use a combination of movement detection and skin-region detection in order to locate people. A combination of these two approaches worked better than either of them on their own, whilst being computationally fast enough to run in real-time. In order to distinguish which people were VIPs, a blob detection algorithm was implemented using a single colour camera to look for the coloured ribbons worn around the necks of VIPs at the conference.

Alfred was able to navigate the room and localise by using wheel encoders and an odometry based motion model. Alfred used a finite state machine to direct his path towards general "guidance points", some of which are predefined based on the room, and some of which are calculated at run

time. Sonar was used by Alfred to be able to detect static obstacles, and the person recognition module was used to detect people, so that Alfred wouldn't bump into things or people. The use of a finite state machine for navigation behaviour appeared to work well according to the experimental analysis, however it was noted that the approach was fairly rigid and other options could be employed in future to make the robot behave more naturally and react better to certain scenarios.

III. Components

Main Node and Integration

The main node, written in Python, operates over the Navigation Stack layer and is responsible for connecting together all the different parts of the system.

Structure

We adopted a modular structure in order to respect the separation of concerns of each part and allow for an easier logging and debugging. The overall system is divided into the following modules:

- Main Node: abstract layer for controlling behaviours.
- Behaviour: responsible for implementing each behaviour interacting with different components.
- Robot: low level layer for communication with external ROS nodes such as the P2OS robot driver, localization, and motion planning.
- Sight: Computer Vision and Object Recognition.
- Hearing: Speech Recognition using Python Speech Recognition[7].
- Speech: Text-to-Speech using gTTS[8].
- Orders: manages and periodically refreshes the database for all orders.

We also made use of a static file named Places to store information about the poses: this holds the exact coordinates and orientation of all the fixed target locations we are using for the various tasks.

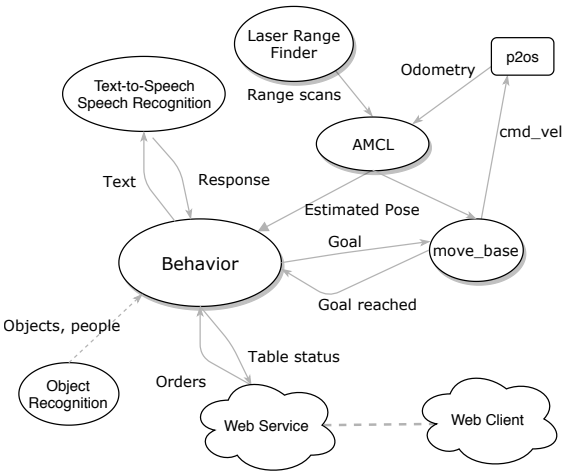

Fig. 2. Components diagram

Requirements

The node requires a Pioneer 3-DX robot with Localization and Navigation Stack running. The following topics are used to communicate with the NavStack:

- /initialpose: publisher access, used to setup the initial pose.
- /move_base_simple/goal: publisher access, used to send goals to the Navigation Stack.
- /move_base/status: subscriber access, used to monitor the current status of the path execution.

An internet connection is necessary to interact with the robot from the web interface and from the Siri vocal shortcut. It is also required when using TTS and Speech Recognition.

Operation Sequence

Initialisation

The first step is initialising all the components, including GPU bindings and topics publishers/subscribers. During this phase the initial pose is set to the chosen starting location.

Loop

The operation loop consists mainly of and update of the orders from the database and three behaviours that are triggered on certain conditions, in the following order of priority:

1) Execute order: collect the required item and bring it to the table.
2) Check for refill: go to a previously served table and ask if the user would like a refill.
3) Wander around: check the tables sequentially for occupancy and waste to be collected, without interacting.

Execute order

This behaviour is triggered whenever one or more orders are available in the queue. The robot will navigate to the pickup location, scan the items, and recognise which products are there, to then ask to be loaded with the appropriate one. The system is designed to keep track of the available products by counting them, and allows the users to only order the available ones. For testing purposes and because of a not enough accurate recognition of the items, we decided to leave this constraint as optional. In fact, the items are being counted and the stock updated, but there is no limitation on the user in ordering an out-of-stock item. The robot will simply ask to be loaded with such item, which can be declined and therefore the order is cancelled. After picking up the item, the robot navigates to the table and, after calling the user by name, delivers the product. It also performs a scan of the table to see if there are more people seated and, if so, asks them if they would like to be served as well. A positive answer will trigger a vocal order creation procedure on the robot and the order will be enqueued.

Check for refill

This behaviour is triggered whenever there are no more orders to execute and and a fixed period of time has elapsed since an order delivery. The operation flow consists of picking the oldest available order for which a refill check is yet to be performed, go to the specified table, take a

scan and ask the person for a refill. A positive response will enqueue an order identical to the original. As for the order execution, the scan of the people sitting at the table is performed so that other possible occupants are asked if they would like to be served. Note that this time if the robot cannot find anyone at the table it will not ask for a refill, assuming the user has left the table. A limitation of this approach is that, without being able to tell apart different users by means of face recognition, we cannot be sure about the person sitting at the table being the same that placed the order in the first place, so our compromise is to just make sure that at least one person is present before interacting. For demonstration purposes only, the refill time window was set to 1 minute, while a more appropriate parameter for a real operation would be 30 minutes.

Wander around

This behaviour is executed when there is nothing else left to do. The robot will simply check each table, by repeatedly visiting the least recently updated. The check consists in visually counting the number of people and of bottles at the table. If everyone has left but some bottles are still present, then they will be labelled as waste and a collection notification will be sent to the cleaning staff dashboard. In this phase the robot is not interacting with the people sitting at the table. This choice was made to not cause annoyance to users and because the wandering behaviour is the default task for the robot, while waiting for a new order or a refill timeout to expire.

Testing

The modular structure of the system allowed us to work simultaneously on different parts, and to exclude some of the modules for testing purposes. In fact, we created dummy modules in order to be able to test single parts faster and isolated from the others, by providing the same interfaces as the real ones, and so making them fully interchangeable. This approach provided us with a full working system where to test single modules, even when offline or with one (or more) of the actual modules not ready. The simulated modules we created include:

- Places: a specific list of places to be used on the StageROS simulation.
- Sight: simulation of Object Recognition with hard-coded results.
- Hearing: simulation of Speech Recognition with keyboard input.
- Speech: local TTS engine.
- Orders: local orders database.

During our tests we faced many situations where an automatic decision was necessary, because of too much noise or no user interaction. We therefore introduced timeouts and default choices for all the user interaction parts. This is particularly evident in speech recognition, where the number of attempts to recognise an answer is limited to three and every recognition has a timeout of eight seconds, which is done to avoid hanging indefinitely. In every case the default option is the one requiring no further interaction (e.g. "No" for a refill request).

LOCALIZATION

For the localization component a custom Particle Filter algorithm was developed in C++ from scratch, and is based on aMCL but uses Sampling-Importance-Resampling rather than KLD-sampling. However, the final implementation of this project uses ROS-provided aMCL node. The configuration of this is explained at the end of this section, after the description of this custom Particle Filter.

The main reason for exploring a different approach and writing everything from scratch was to have the filter obtain a better pose proposal distribution, by using both the odometry and the laser scan measurements during the prediction step. This is done by performing Iterative Match Point-Range[9] (IMRP) on the two laser scans with RANSAC outlier rejection[10], as it performs better than Iterative Closest Point (ICP) in cases of large rotational error, which is more likely with a differential drive robot such as ours. The key idea behind scan matching is to estimate the rigid-body transformation between the scans z_{t-1}, z_t taken at two poses x_{t-1}, x_t. This is accomplished by running multiple iterations of scan matching using samples from the prediction (which incorporates the odometry reading u_t) as initial guess to this transformation, therefore sampling from $p(x_t|x_{t-1}, u_t)$. RANSAC is used to reject outliers while MeanShift clustering is applied to the residuals of the matching for each sample, giving a new set of samples over $p(x_t|x_{t-1}, u_t, z_t)$.

The resampling is triggered by a drop in the Number of Effective Particles or N_{eff}. This number is approximated as $\frac{1}{\sum_{i=1}^{N} w_i^2}$ or the inverse of the sum of squared weights, which are assumed to be normalized to sum to 1. This is an approximation to the weights' variance, as when all weights are equal ($\forall i \; w_i = \frac{1}{N}$) $N_{eff} = \frac{1}{N\frac{1}{N^2}} = N$ meaning all particles are contributing to the pose estimate as they are equally likely. Conversely, in the degenerate case, when most particles' weights tend towards zero and only a few particles are likely, N_{eff} will tend to 1. When N_{eff} drops below a specified threshold, which we chose to be $\frac{N}{4}$, the resampling is performed. Resampling is performed by means of Sampling-Importance-Resampling [11](SIR), which will filter out unlikely particles and improve the sample distribution by computing importance weights, which is calculated as the ratio between the proposal and target distribution. The normalization of the weights is given by $w_i = \frac{w_i}{\Sigma_{i=1}^{N} w_i}$, and ensures that they sum to 1. This is done after both the prediction and the update/resampling step.

Due to the additional complexity of this filter from the scan matching component with outlier rejection, it was not possible to finished development and testing within the time frame assigned. The scan matching often diverged, compromising the overall stability of the filter. Therefore even though aMCL uses a simpler proposal distribution estimation, we managed to get good results for both normal localization and robot kidnapping.

The main global parameter is the valid interval for the amount of particles representing the distribution, which we set to be between 50 and 1000, since we have a large map and noisy odometry, requiring many particles to cover

sufficiently. Another important global parameter is that of the minimum movement between updates, which ensures enough difference with the previous pose: this was set to 0.1 meters and $\frac{\pi}{24}$ radians (15°). This is to prevent the filter from running multiple times in locations very close to each other, which can happen in case of slow or in-place movement.

The recovery alpha slow/fast parameters adjust the exponential decay rate that adds random poses to the distribution estimate. This is of significant importance when considering the kidnapped robot problem: adding random poses helps recovering from a bad estimate, as one of the added poses may lie close to the true state, giving the filter the opportunity to assign an increasingly higher importance to that sample, if it is evaluated to be more likely than the others. These values were set to 0.005 and 0.2 for slow and fast recovery alpha parameters respectively.

The parameters for the motions model represent the expected noise in the translational and rotational components of the odometry, and were left to the default values of 0.2. Increasing them leads to a more unstable (and in some cases divergent) filter, while reducing them prevents it from propagating the particles far enough to ensure a good portion of them is close to the true underlying distribution.

The sensor model is parametrised on the probability of a beam corresponding to the ray-traced value, and it is configured based on our sensor's specifications (URG 04LX)[12]. The sensor is capable of measuring the range to obstacles within 5.6 meters and in the range $\pm$ 120°at a rate of 5Hz, providing around 683 readings at an angular resolution of 0.352°. The error in the measurements of our laser scanner is 3 centimeters, or 3% of the distance if this greater than 1 meter. The sensor model replicates this on the map by means of ray-tracing, although at an angular resolution of 1°, and is used in the update step (z_t) to assign an importance weight to each particle.

The parameters for the sensor model were set to the following:

- laser_max_beams: 360, represents the number of beams to ray-trace
- laser_z_hit: 0.90, represents the probability of a beam responding correctly to the presence of an obstacle
- laser_z_short: 0.2, represents the probability of a dynamic (not in the map) obstacle preventing the measurement of the real obstacle behind it and therefore returning a range that is much shorter than expected
- laser_sigma_hit: 0.075, represents the standard deviation (or noise) in the reading when the beam reaches an obstacle
- laser_lambda_short: 0.05, represents the exponential decay in the reading when the beam is cut off by an unknown dynamic obstacle

The main difference with the default parameters is that we found that increasing the probability and reducing the lambda parameter of short hits improved the response when a good part of the environment is occluded from the laser scanner, which is often the case when many dynamic object, such as people walking by, are present.

NAVSTACK

For the Motion Planning and Execution we used the ROS NavStack, using TrajectoryPlannerROS with Dynamic Windows Approach (DWA) for local planning. Other available planners such as Elastic Band (EB) and Timed Elastic Band (TEB) were explored, however the former often gets stuck in dynamic environments as it is subject to local minima capturing, while the latter is much more computationally expensive than DWA without any substantial increase in path quality.

The main principle of operation of this planner is to first generate a global plan as a sequence of valid positions in the global costmap, then at each time step use the local planner to create a set of plans near the global path, within a certain window. The size of the local costmap window was chosen to be 5 meters in both width and height, as this measure is similar to the maximum detectable range of our laser scanner and allows for the simulation to evaluate collisions with nearby obstacles. The resolution of this costmap was chosen to be of 0.05 meters: since range measurements have an error of similar magnitude (minimum 3cm), setting a lower value would only waste computational effort, whereas a higher value leads to more jagged movements. These plans are then evaluated by forward simulation and the one with lowest overall cost is picked. This process is repeated at a rate which was decided to be the same as the laser scanner updates, which is the source of obstacle range information from which the local costmap is built. This rate is of 5Hz, which means a time slot of 0.2 seconds.

For the tuning of the NavStack parameters, a very useful reference was the "ROS Navigation Tuning Guide" found in [13].

Costmap parameters

A costmap is a data structure that holds the values for a cost function in the form of occupancy grid (matrix of cells which correspond to coordinates in the map). The cost is a value in the range 0-255 and represents the penalty of navigating through that cell. The global costmap is generated by taking the complete map of the environment, and assigning the lethal cost value (i.e. 255) to the obstacles found in it, and is accounted for in the global planning. The local costmap is instead a window centered on the robot's pose, and is updated with each incoming laser scan by a plugin called Obstacle Layer. The update consists of running two procedures: the marking update will place obstacles seen in the laser scan (within a distance defined by the obstacle range parameter) into the layer, whereas the clearing update will clear the costmap along the ray-trace from the observation source's center to the obstacle (within a distance defined by the ray-trace range parameter). The obstacle layer is also paramterized on the robot footprint, which in our case was set to be circular with a radius of 0.3 meters.

The other plugin used on the costmaps is the Inflation Layer: this propagates the cost of each obstacle to nearby cells within a specified radius, with a value inversely proportional to both the distance from that obstacle and the cost scaling factor. A high factor will lead to a fast drop in cost over distance, whereas a lower value will create a

more gradual slope, as well as providing a better response in more constrained environments such as narrow passages. Experimenting with these values showed that having a softer boundary between free and occupied cell helps the local planner formulate a good path by default, because it will try to keep the path as equidistant from obstacles as possible. On the other hand, if the radius is too small, the planner would perceive being in the center of a hallway and being very close to a wall in the same way, and therefore will produce inadequate plans.

The global costmap is added an Inflation Layer with radius 0.2 meters and cost scaling factor of 2.5, to ensure that the global planner will never formulate plans residing too close to walls and other static obstacles. For the local costmap however, we found that good values to obtain safer paths as described above were with a radius of 2.5 meters, and a scaling factor of 1.25. This allowed the robot to achieve smooth and conservative movements even in presence of many dynamic obstacles such as people walking by.

Global configuration and paramters

The ROS NavStack has many parameters and plugins that can be configured. The main parameters regard the movement constraints, and the Pioneer 3-DX robot is not capable of holonomic motion. Other global parameters that control the planning include physical limits such as speed and acceleration, as well as controls affecting path quality such as forward simulation time and its granularity, and the goal tolerance. The upper bound for physical limits are determined by the robot, however for both speed and acceleration we chose much smaller values to obtain a safer behaviour and give us time to react in case of failures. These limits were set to a range of 0.15-$0.5ms^{-1}, 0.4$-$1.0rad\ s^{-1}$ for speed and $0.3ms^{-2}, 0.5rad\ s^{-2}$ for acceleration, for linear and rotational components respectively. The goal tolerance was set to $0.2m, 0.2rad\ s^{-1}$ for linear and rotational components of the goal pose, so to allow the robot to reach a destination even when the exact location is occupied. In fact, the planner will not emit a path when the current local goal resides in the local costmap but is unreachable (due to obstacles).

The simulation parameters can severely affect performance, as they are a trade-off of runtime and path quality. The forward simulation time adjusts how far in time the planner should evaluate each plan, while the granularity controls at what interval the path should be sampled. The parameters are such that the simulation is performed over 3 seconds at a granularity of 0.1, giving 30 evaluations steps for each trajectory. Other important simulation parameters include the number of samples to take in the state space when generating local plans, which is also a performance trade-off. The values of 10 and 15 samples for translation and rotation respectively were found to lead to good plan quality without overrunning the planner time slot of 0.2 seconds. As the Pioneer 3-DX can only move along its x-axis, the number of samples in the y configuration space was set to 0, as lateral translation is achieved by altering the heading.

Recovery behaviours

A feature of the ROS planner is that of fallback to other behaviours whenever a plan cannot be found within a timeout, defined by the planner patience parameter. These behaviours consist of actions that are expected to improve the chances of finding a valid plan. The first behaviour is called conservative reset, and clears the persistent obstacles in the local costmap that are no longer visible by the laser scanner. This allows the robot to move to locations that resulted occupied in an earlier observation. If this doesn't succeed, the robot will perform a recovery rotation. This consists of slowly spinning the robot in place, and allows for the laser scanner to take a complete measurement of the surrounding obstacles. This can help the localisation improve the pose estimate while also updating the costmap. If this is not sufficient, the local costmap will be erased by the aggressive reset recovery behaviour, and the planning will start from scratch. If the planner fails again then it will halt, and the goal will be considered unfeasible.

OBJECT RECOGNITION

The object recognition part of this project is using the YOLO Real-Time Object Detection[14] library combined with the Tensorflow Image Classifier[15] that helps to categorise the bottles that the robot will deliver.
How YOLO works:
It applies a single neural network to the full image to predict the bounding boxes surrounding each recognised object, and provides confidence scores for each. These bounding boxes are weighted by the predicted probabilities and finally only the high scoring predictions will be selected.
YOLO configuration:
YOLO has multiple pre-trained models that offer different levels of accuracy and frames per second performance (FPS). For the purposes of our project we choose to use yolov2 with the Darkflow[16] implementation.

The bottle classification uses the Tensorflow Image classifier which we trained to recognise our own images. Our training set is formed of approximately 700 images for each of the 3 classes that we used (Coca Cola, Coca Cola Light, Water). The two systems that we described above are combined and all the images that are classified as bottles by YOLO are fed into the Tensorflow Image Classifier, which will assign one of the 3 labels mentioned above.
Development stages:
For the bottle classifier we have tried multiple approaches. The first one was using the most predominant colour in the picture: this technique was implemented by using K-Means clustering which would output the largest cluster representing the dominant colour. However, even if we made the assumption that each bottle will have a different colour this method didn't yield good results because of the large computations necessary for clustering it would decrease the FPS significantly, as well as being highly sensitive to illumination variance. The second approach used a random forest classifier instead of the K-Means clustering which required less computing, as the model was trained beforehand. This method showed a good performance when tested on the validation set but on the real-time detection achieved poor

results due to the ambient variations. In order to deal with this problem we used the TensorFlow Image Recognition which is built to achieve good performance on difficult visual recognition tasks because it uses a Convolutional Neural Network and it is trained on ImageNet. After retraining it on our dataset mentioned above, the classifier scored an accuracy of over 80% on the testing set and satisfactory performance on the real-time detection. The output of the classifier can be seen in Figure 3.

Fig. 3. Bottles Classification

WEB INTERFACE

The waiter robot web UI has been designed with a Mobile-First approach, that is an important process when creating a website. This design strategy aims to start sketching and prototyping the smallest screen first, and then work the way up to larger screens, leading to ensure the right user experience for the right device.
For the interfaces, we chose to use AdonisJs [17], a framework that uses Edge and Nodejs for the front and back-end of the system respectively.

NodeJs

Nodejs [18] is a JavaScript runtime environment that executes code outside of a browser; it is built upon Chromes V8 JavaScript engine. Nodejs lets developers use JavaScript to write Command Line tools and server-side scripts to produce dynamic web page content to be then sent to the user's web browser. Nodejs uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, which makes it very popular across various popular websites.

Edge

Edge [19] is a template engine that follows the natural JavaScript syntax, which makes it easier to type and remember. Using Edge gives us the possibility to have Conditionals, Iterations, Components and Partials. For example for string interpolation, Edge templating engine makes use of mustache syntax {{username}}.

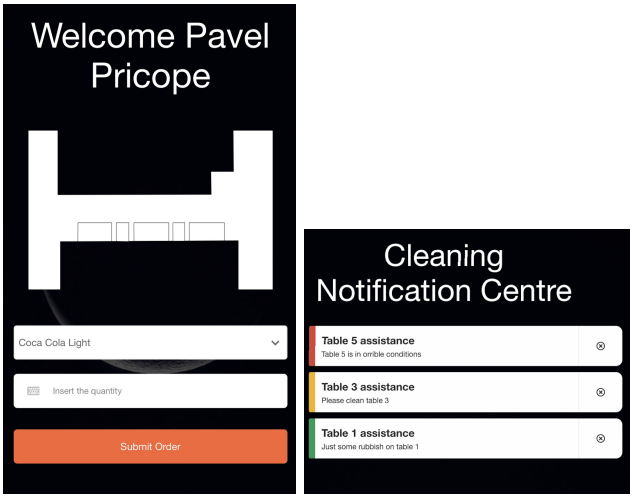

Fig. 4. UI

Web Interface features

The back-end of the system has been configured to manage multiple role permissions and actions using an ACL (Access Control Lists) for database authentication. During the login phase, the system is going to identify the role of the profile attempting the login, and by means of a Routing System it alters the endpoints (URIs) it responds to. The two ACL profiles that we choose to create are App_User and Service. The behaviour of the server on the bases of these two roles are:

- App_User: The web page that the server provides for this profile is a web view where the user can place an order asking the robot to bring a specific product into a location chosen on the map, as can be seen on the left frame at Fig. 4.
- Service: This profile is intended to be used from the University of Birmingham cleaning service. In case of login using this profile, the server is going to render a view that allows the cleaning service to receive notifications for a necessary service intervention as can be seen from the right frame at Fig. 4.

IV. RESULTS

The evaluation of the results was conducted on these two scenarios and later extended to a more comprehensive operation.

- Order a drink to a table: A user will visit a web application and will order some items from the menu. The robot will then navigate to the pickup location where it will look for the object, and once identified it will ask someone to be loaded with such object. Then, it will find its way to the table and deliver the order. After a 30 minutes time, the robot will go back to the table asking for a refill, and upon a positive response will place and execute another order, identical to the original one. (Note: time reduced to 1 minute for testing purposes)
- Wandering: The robot will wander around the tables and will try to identify bottles and people sitting there. If at least a bottle is detected and no one is sitting

at the table, then the robot will identify the bottle(s) as leftovers, sending a notification to the cleaning services.

Order a drink

Preconditions:

- Robot: initialised with initial pose.
- Orders: empty queue. No past order.

Actions taken:

- An order will be placed at Table 4, after receiving the product the user will wait for the refill check, and give a positive answer.
- Two people will seat at the table, but after the first order delivery one of them will leave.

Observed sequence:

- Wandering behaviour is triggered since there is no order in the queue and no refill to operate on previous orders.
- Robot wanders to the first table. Since no table has been visited before, this is Table 1.
- An order for 1 Coca Cola Light at Table 4 is sent by a user called Pavel.
- After checking Table 1 (empty, no leftovers, no action) the robot picks up the order and navigates to the pickup point.
- An obstacle is on the way at the centre of the corridor. The robot passes it on the right and no recovery behaviour is triggered.
- The robot reaches to the pickup location with an acceptable precision.
- The robot asks to be loaded with 1 Coca Cola Light, a user confirms the operation via keyboard.
- The robot navigates to Table 4 and takes a scan of the environment (2 people present, 2 people found).
- The robot says "Pavel, here is your Coca Cola Light", action is confirmed via keyboard.
- The robot asks if anyone else wants a drink.
- A negative answer is given.
- The robot drives away, wandering behaviour is triggered and the next table is Table 2 (Table 1 already visited, Table 4 just visited).
- After the refill timeout for this order expires, the robot navigates back to Table 4 and counts the number of people. At least one person is detected and the robot says: "Hi Pavel, would you like a refill?"
- Pavel answers positively.
- An order is created (duplicating the original one) and because the queue was empty and the refill task is complete, the new order is immediately picked up by the robot.
- The robot navigates to the pickup location. A correct alignment is reached.
- The robot asks to be loaded with 1 Coca Cola Light (the refill). Operation again confirmed by keyboard.
- The robot navigates to Table 4. An obstacle is detected an a re-routing is performed correctly.
- The robot takes a scan of the environment (1 person present (1 has left), 1 person found)
- The robot says "Pavel, here is your Coca Cola Light", action is confirmed via keyboard.
- No further question is asked since no one is at the table except Pavel.
- The wandering behaviour is triggered.

Verified hypothesis:

1) An order with refill is correctly executed.

Wandering

Preconditions:

- Robot: initialised with initial pose.
- Orders: empty queue. No past order.

Actions taken:

- A bottle will be placed on Table 1, with no people seated.

Observed sequence:

- Wandering behaviour is triggered since there is no order in the queue and no refill to operate on previous orders.
- Robot wanders to the first table. Since no table has been visited before, this is Table 1.
- Correct alignment with Table 1 is reached, and a scan is taken (0 people present, 0 people detected - 1 bottle present, 1 bottle detected).
- A bottle is identified and because no person is detected, the robot sends a notification to the cleaning dashboard.
- A notification appears saying Table 1 has some rubbish, severity is low (1 bottle).
- The wandering behaviour is triggered again.

Verified hypothesis:

1) During a wandering the robot correctly discovers a leftover bottle and reports it to cleaning service.

V. Conclusion

This section summarises the overall conclusions drawn from this project and describes how it could be improved upon with further work. The fundamental conclusions that can be made are:

- An autonomous system capable of object recognition that acts as a waiter for the people using the study spaces has been successfully created. The system is fully able to execute all the objectives set at the beginning.
- The custom implementation for the particle filter was promising, and if explored in more depth it could lead to a very good localisation solution. More recent localisation approaches such as those based on graph-SLAM (e.g. iSAM2, g2o) could have been considered, however they may require more effort for the parameter tuning.
- The further development of a custom NavStack using our A-Star node could lead to a more confident-looking navigation, as the search algorithm can be highly customised for this specific robot and use-case scenario. However, in order to obtain a full NavStack replacement, a local planner would also need to be developed, either by using A-Star for this as well or by implementing a simpler algorithm to perform collision avoidance.

- The Image recognition is one of the components that could be improved upon by using a more comprehensive dataset which would make the system less vulnerable to changes in ambient lighting.
- Facial recognition is a feature that could be added and would be useful when the robot asks previous clients for a refill, as it would ensure that it is asking the correct person.
- The web interface can be improved in multiple aspects to offer a production-ready solution. The users could be registered using the University account, restricting the access to the system to be University only. The system can be extended to support a more vast environment, offering a third role for the administrative staff to update the system settings. It could also be expanded to support multiple floors or different buildings, rerouting the request to the appropriate robot.

REFERENCES

[1] Dieter Fox. Adapting the sample size in particle filters through kld-sampling. 2002. http://www.robots.ox.ac.uk/~cvrg/hilary2005/adaptive.pdf.

[2] S. Engelson and D. McDermott. Error correction in mobile robot map learning. 1992. https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=220057.

[3] M Asif, M Sabeel, and Zeashan Khan. Waiter robot solution to restaurant automation. 11 2015.

[4] Bruce Maxwell, Lisa Meeden, Nii Addo, Laura Brown, Paul Dickson, Jane Ng, Seth Olshfski, Eli Silk, and Jordan Wales. Alfred: The robot waiter who remembers you. *Ai Magazine - AIM*, 12 2018.

[5] Byron Reeves and Clifford Nass. The media equation: How people treat computers, television, and new media like real people and places. *Bibliovault OAI Repository, the University of Chicago Press*, 01 1996.

[6] Viavoice. http://support.nuance.com/usersguides/default.asp.

[7] Python speech recognition. https://pypi.org/project/SpeechRecognition/.

[8] Gtts. https://pypi.org/project/gTTS/.

[9] Feng Lu and Evangelos Milios. Rob ot pose estimation in unknown environments by matching d range scans. 1997. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.43.4126&rep=rep1&type=pdf.

[10] Chieh-Chih Wang Shao-Wen Yang and Chun-Hua Chang. Ransac matching: Simultaneous registration and segmentation. 2010. https://www.csie.ntu.edu.tw/~bobwang/Papers/yang_icra10.pdf.

[11] Kari Heine. Unified framework for sampling/importance resampling algorithms. 2005. http://math.tut.fi/posgroup/heine_fusion05a.pdf.

[12] Hokuyo. Urg-04lx-ug01 technical specifications. https://www.hokuyo-aut.jp/search/single.php?serial=166.

[13] Kaiyu Zheng. Ros navigation tuning guide, 2016. http://kaiyuzheng.me/documents/navguide.pdf.

[14] Yolo. https://pjreddie.com/darknet/yolo/.

[15] Tensorflow image recognition. https://www.tensorflow.org/tutorials/images/image_recognition.

[16] Darkflow. https://github.com/thtrieu/darkflow.

[17] About adonisjs. https://adonisjs.com/docs/4.1/about.

[18] About node.js. https://nodejs.org/it/about/.

[19] Node.js templating engine with fresh air. https://edge.adonisjs.com/.

APPENDIX

aMCL Parameters

```
recovery_alpha_slow: 0.005
recovery_alpha_fast: 0.2
laser_max_beams: 360
laser_z_hit: 0.95
laser_z_short: 0.2
laser_sigma_hit: 0.075
laser_lambda_short: 0.05

update_min_d: 0.1
update_min_a: 0.15

min_particles: 50
max_particles: 1000
```

Local Planner Parameters

```
controller_frequency: 5.0
TrajectoryPlannerROS:
  escape_vel: 0.05
  max_vel_x: 0.5
  min_vel_x: 0.15
  max_vel_y: 0.0
  min_vel_y: 0.0
  max_rotational_vel: 1.0
  min_in_place_rotational_vel: 0.4
  acc_lim_th: 0.5
  acc_lim_x: 0.3
  acc_lim_y: 0.0

  holonomic_robot: false
  yaw_goal_tolerance: 0.2
  xy_goal_tolerance: 0.2
  goal_distance_bias: 0.8
  path_distance_bias: 0.6
  sim_time: 3.0
  sim_granularity: 0.1
  heading_lookahead: 0.4
  oscillation_reset_dist: 0.02
  conservative_reset_dist: 1.0
  pdist_scale: 0.4
  publish_cost_grid_pc: true
  global_frame_id: "odom"

  vx_samples: 10
  vy_samples: 0
  vtheta_samples: 15
  dwa: true
  planner_patience: 15.0
```

Common Costmap Parameters

```
obstacle_layer:
    obstacle_range: 2.5
    raytrace_range: 5.0
    robot_radius: 0.3

    observation_sources: base_scan
    base_scan: {sensor_frame:
       base_laser_link, data_type:
       LaserScan, topic: base_scan,
       marking: true, clearing: true}

inflation_layer:
  enabled:              true
  cost_scaling_factor:  2.5
  inflation_radius:     0.2

recovery_behaviors:
    - {name: conservative_reset, type:
       clear_costmap_recovery/
       ClearCostmapRecovery}
    - {name: rotate_recovery, type:
       rotate_recovery/RotateRecovery}
    - {name: aggressive_reset, type:
       clear_costmap_recovery/
       ClearCostmapRecovery}
```

Local Costmap Parameters

```
global_costmap:
  global_frame: /map
  robot_base_frame: /base_link
  update_frequency: 5.0
  publish_frequency: 2.0
  static_map: true
  transform_tolerance: 0.5
```

```
  plugins:
     - {name: static_layer, type: "
         costmap_2d::StaticLayer"}
     - {name: obstacle_layer, type: "
         costmap_2d::VoxelLayer"}
     - {name: inflation_layer, type: "
         costmap_2d::InflationLayer"}

  inflation_layer:
    enabled:              true
    cost_scaling_factor:  2.5
    inflation_radius:     0.2
```

Local Costmap Parameters

```
local_costmap:
  global_frame: /odom
  robot_base_frame: base_link
  update_frequency: 5.0
  publish_frequency: 5.0
  static_map: false
  rolling_window: true
  width: 5.0
  height: 5.0
  resolution: 0.05
  transform_tolerance: 2.0

  plugins:
    - {name: obstacle_layer, type: "
        costmap_2d::VoxelLayer"}
    - {name: inflation_layer, type: "
        costmap_2d::InflationLayer"}

  inflation_layer:
    enabled: true
    cost_scaling_factor:  1.25
    inflation_radius:     2.5
```