Robotic Systems Report

CJLZ-2
Sergio Castillo-Hernández(jc15275)
Vince Jankovics (vj15292)
Miguel Lagunes-Fortiz(ml15765)
Kaiyang Zhou(kz15291)

1 Introduction

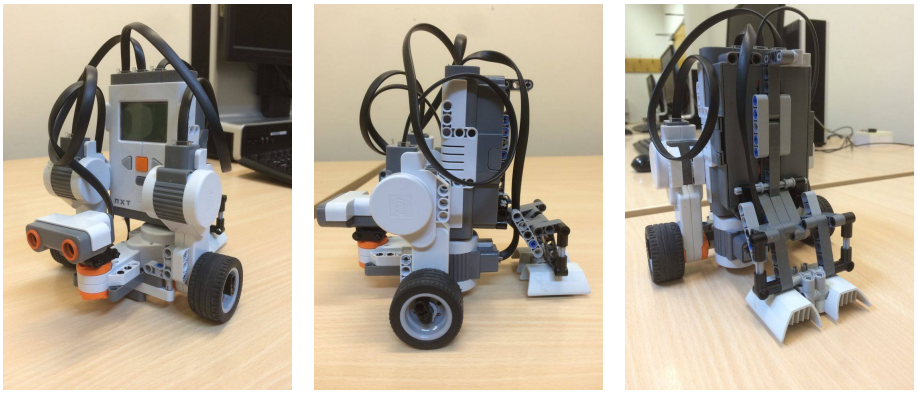
"Kidnapped the robot" is a test problem used commonly in the mobile robotics field. This test determine how accurate a robot can estimate its position given a map, random start point and a target destination, then the robot using sensors have to measure its environment and calculates its position.

This report will cover how we attacked this problem and how we solved it. It is divided in [add number of sections]. It will first cover the design and assembly of the robot. It will then describe our solution to the localisation, navigation and path planning. The third part provide the results obtained from both, the simulation and the real robot. Finally, some discussion about the difference between simulation and the real robot found while conducting our test.

2 Robot Design

We mainly focus on two aspects when configuring the Lego robot, namely *I)* stability and *II)* feasibility. The complete configuration of the robot is visualised in Fig.1, where there are three images showing the front, side and back appearances of the robot respectively.

The total height of the robot is approximately 18 cm. The width and length are 13 cm and 19.5 cm, respectively. This configuration ensures that the robot does not occupy too much space in the arena. The ultrasonic sensor is placed in the lower front with an approximate height of 6.5 cm and has a vision range of 180 degrees. A previous prototype with the sensor on the top and 360 degrees of freedom was tested but the wire connected to de brick interfered the sensor spin. A novel feature used in this design is the support device installed in the back as shown in Fig.1c, where two balls are positioned inside the two white slots to reduce the friction between the ground and the support device while rotating and moving.



(a) Front appearance (b) Side appearance (c) Back appearance

Figure 1: Robot configuration.

3 The kidnapped robot problem

3.1 Navigation

In this section, we provide detailed implementation of particle filters localisation (PFL) and novel features used to boost the performance. The Bayesian reasoning of particle filters, however, is not focused in this section.

Algorithm 1: Particle filters localisation

```
Input: particles, robot, isConverge
Output: estimate, isConverge
updateParticles(particles, robot);
estimate = getEstimation(particles);
if isConverge = true then
    if estimate is too different from real robot then
        initialise particles;
        isConverge = false;
    end
else
    isConverge = checkConverge(particles);
end
return estimate, isConverge
```

We determine the size of particles as 400 via comprehensive experiments, which will be explained later. The implementation of PFL is demonstrated in Algorithm 1. The input to this algorithm includes 400 particles, the real robot and a parameter called *isConverge* representing the status of these particles. When *isConverge* equals to 1, these particles are converged and 0 otherwise. The output of this algorithm contains two elements: first is the estimate of the real robot that contains estimated position and orientation; second is the status parameter that commands the robot to continue in exploring the environment to find its location or to move towards the target position.

Initially, the particles are positioned randomly inside a map. We add movement noise and turning noise to the particles using the given functions in the BotSim library. During the localisation process, we first update the particles in terms of their positions and orientations, which is enclosed in the function *updateParticles()*. To do so, firstly we obtain the scans (sensor readings) of the robot and the particles. These scans are represented as vectors $\mathbf{r}$ and $\mathbf{p}_i$, where $i = 1, ..., 400$, for the robot and particles respectively. Secondly, to update each particle's weights, we calculate the difference between the scan of the robot $\mathbf{r}$ and the scan of each particle $\mathbf{p}_i$ using the $\ell 1$-norm i.e. $d_i = ||\mathbf{r} - \mathbf{p_i}||_1$, and then obtain the new weight for each particle by

$$w_i = exp(-\frac{d_i}{2\sigma^2}) + damp$$

where *damp* is a damping factor used to prevent the weight from being zero. We use $\ell 1$-norm instead of the $\ell 2$-norm to compute the difference between two sensor readings because using $\ell 2$-norm will make the weights of the particles less informative. For example, when using the $\ell 2$-norm, each d_i becomes much bigger since the difference is squared

for each pair in the two vectors (sensor readings) and accordingly $exp(-\frac{d_i}{2\sigma^2})$ becomes much smaller and most of them are close to zero. As a consequence, all of the weights of the particles are nearly equal to the dampling factor, which makes the robot harder to match a particle. The σ and *damp* are assigned 3 and 0.01 respectively, which are obtained by running lots of experiments and selecting the optimal setting. Once the weights are computed, we normalise them by

$$w_i = \frac{w_i}{\sum_{i=1}^{400} w_i} + \alpha$$

where $\alpha = 0.5/400$ is a constant. The reason for adding this constant will be explained later when we describe our resampling process.

The particles with larger weights are closer to the real robot. To do the resampling, we first resort the particles in a weight-decreasing order so the particle with the largest weight is positioned at the first and then we redistribute these particles. The particle with the max weight is copied for $n_i = w_i \times N$ times where $N = 400$, and we repeat the same step for each particle in the queue until the number of the new particles is equal to 400 i.e. $\sum n_i = 400$. However, during this process, we found that the number could not reach to 400 because many n_i equaled to zero since $w_i \times N < 0.5$ (we used round to get an integer). Therefore, we add the constant $0.5/N$ to ensure the resulting n_i satisfies $n_i \geq 0.5$ such that n_i can be rounded to 1 rather than 0.

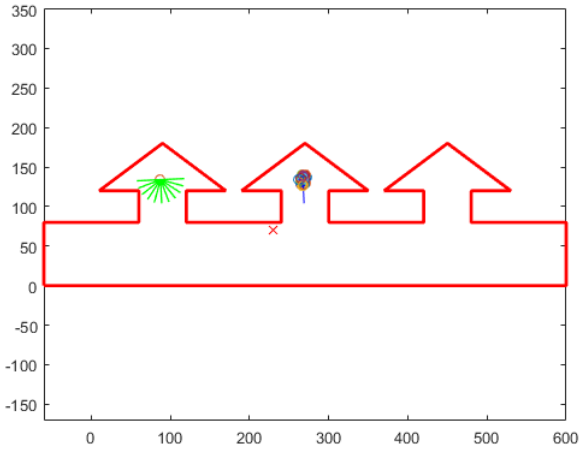


Figure 2: Robot in a symmetric environment causing the symmetric ambiguity problem.

The *estimate* of the real robot that contains the position and orientation information is computed by separately averaging the positions and orientations of the new particles. After this, we check whether the particles were converged previously i.e. we go to Line 3 in Algorithm 1. There are two situations that will happen: (**1**) If they were converged, we need to check whether the obtained *estimate* matches the real robot or not. This step is necessary because if the map is somewhat symmetric as shown in Fig.2, the particles were very likely to converge in a wrong place although the sensor readings were consistent with the robot's. We compare the *estimate* with the real robot in this way: suppose the sensor readings of the *estimate* are **e** and those of the real robot are **r**, the difference between

them is computed by $df = ||\mathbf{e} - \mathbf{r}||_1$. If df is bigger than a threshold, which means the *estimate* is inaccurate and the particles converged in an incorrect place, all the particles will be randomly positioned again and the status parameter *isConverge* will be set to *false*. (**2**) If the particles were not converged, they need to be checked by the function *checkConverge()*. With the observation that if the particles are converged, they will look compact in the 2D map. We thus use a statistical method to check the convergence: we calculate the 2 by 2 covariance matrix (CovMat) based on the positions (x, y) of the particles and obtain the corresponding eigenvalues of the CovMat; these two eigenvalues describe the spread of the particles and how tight they are; we define the particles being converged if and only if the sum of the eigenvalues is lower than an experimentally determined threshold. By performing this localisation process, the particles are able to accurately find its location in a known map and the symmetric ambiguity is possible to be eliminated.

In order to select an optimal value as the size of the particle filters, we performed substantial experiments in which the size ranged from 100 to 1000 with an increment of 100. Thus there were totally 10 candidate values. We fixed the starting position of the real robot at $(20, 20)$ and the target position at $(80, 80)$ using the first map provided as shown in Fig.**??**. The distance between these two positions is sufficient for the particles to get converged. To judge the performance of the PFL with different size settings, we designed three criteria: *distance error* (e), *convergence time* (t) and *standard deviation* (s). The definitions of these three criteria are specified as follows:

- Distance error (e): it is defined as the Euclidean distance between the target position and the final robot position.
- Convergence time (t): it calculates the time (in seconds) that the particles require to get converged.
- Standard deviation (s): it defines how accurate the estimated position is after the particles get converged and can represent the stability of the algorithm. The mathematical expression is as follows: suppose after the robot finds its location it requires N steps to get stopped, denote the 2D position of the robot at each time step i as P_i^r and the estimation as P_i^e, the standard deviation s is computed as $s = \sqrt{\frac{1}{N}\sum_{i=1}^{N} ||P_i^r - P_i^e||^2}$.

For each size setting, we ran 100 experiments where we computed the e, t and s for each and finally we averaged them as the output. The results are plotted in Fig.3a, where the lower the values are, the better the performance is. It can be seen that the convergence time (green) increases as the size of the particle filters increases. On the distance error (blue) and the standard deviation (red), the algorithm with sizes from 300 to 1000 produces comparable results. Therefore, from this graph we could not decide which size is the optimal. To overcome this problem, we combined these three criteria to form a penalty-based criterion that can be used to distinguish one with the relatively best performance from the rest. We defined this criterion as *penalty-based score*, which was computed by

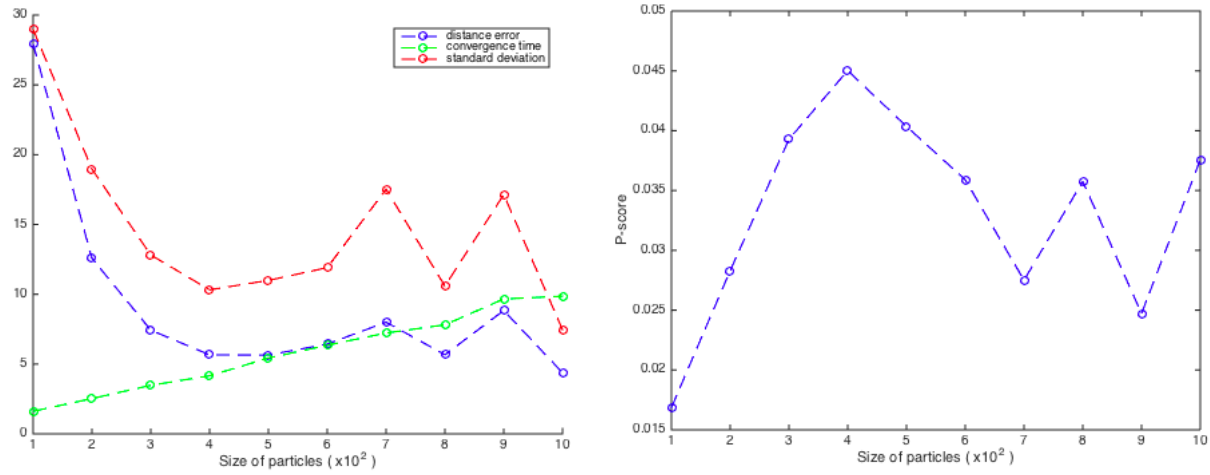
$$psc = \frac{1}{e + 1.5t + s}$$

The constant added before the t aims to increase the weight of the convergence time such that the size that causes longer time for the particles to converge is less wanted since it

decreases the score. As a result, the new ranking is more clear and the size equal to 400 has the best score as shown in Fig.3b. Thesefore, we selected 400 as the optimal size.

By running this algorithm to localise the robot in the simulation, we obtained promising results where the average distance from the target was always less than 3. Here we summarise the novel features used in this algorithm:

- $\ell 1$-norm based weights updating function.
- the α constant that is added when normalising the weights for the particles.
- a re-validation method to measure whether the obtained estimate is correct or not.
- a statistical method to determine the convergence condition.
- a penalty-based score to help select the optimal size for particle filters.



(a) (b)

Figure 3: Experiments to select an optimal size of particle filters.

After finding the optimal number of particles, we aimed for finding an adequate number of beams for the sensor reading towards an accurate and fast implementation.

On the simulation, we let the robot localise itself into the first map and we measured the convergence time of the particle filter and the distance error relative to the actual robot's position, we vary the number of beams from 5 to 40 using increments by 5 and running each experiment 500 times.

We present the results on (Figure 4) and we inferred that more sensor beams leads to a more accurate pose's estimation but also increases the computational time for convergence since more operations are required. Thus, we select 10 beams as an adequate value since offers both accuracy (mean error of 1.44cm) and fast computation time (mean 3.34 seconds for convergence). Thus we use 10 beams for further experiments and implementation.

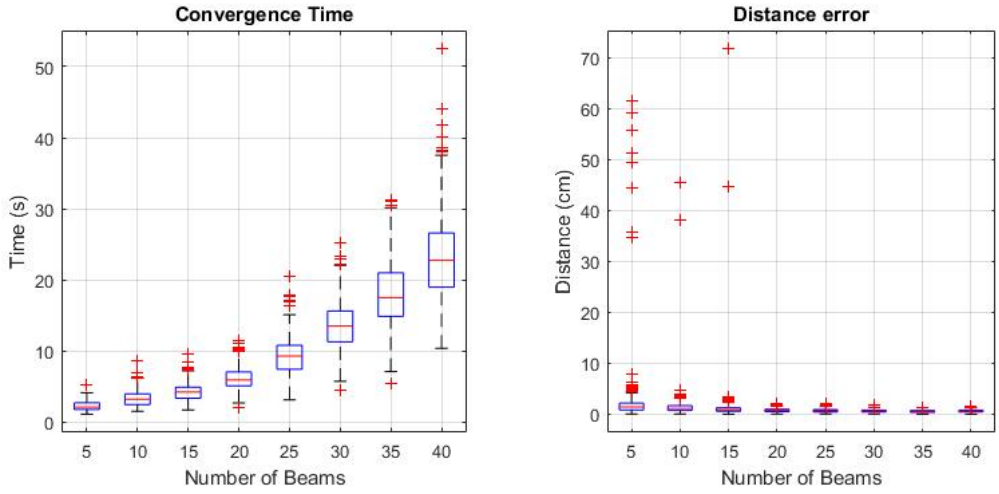


Figure 4: Selecting an adequate number of sensor beams

3.2 Guidance

Before and after the robot is localized in the map, it has to know what to do next according to the current objective, which are:

1. Avoid collision all time,
2. Explore map,
3. Go to target.

3.2.1 Collision avoidence

The robot's highest priority objective is avoiding collisions with the walls. This is done using two main features, re sampling the map and a 'bounce back' algorithm.

Before exploring or planning, a map resize is conducted. This function receives the map coordinates and the distance desired between the wall and the robot. New coordinates are returned, this new map is used in the exploring and path planning steps.

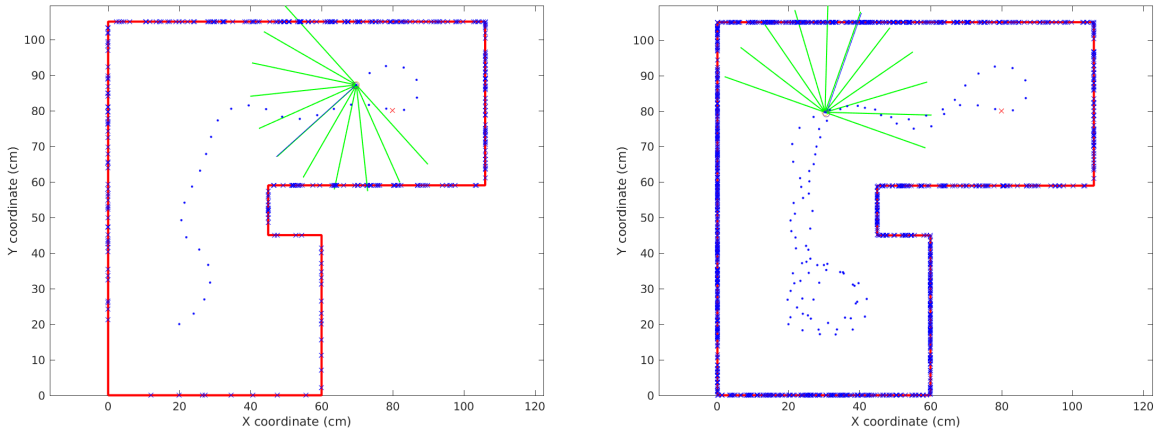
The algorithm estimates the smallest distance between the robot and the scanned points based on the commanded movement. If this predicted distance is smaller than a certain threshold, it 'bounces back' from the boundary. The reflected angle is perturbed with a small noise factor, so the robot does not get stuck in an area (e.g. when it goes perpendicular to the wall).Both features can be seen on Figure 6a.

3.2.2 Exploration

The exploration is done before the PFL is converged to a location (i.e. the localization is not complete), so the robot can wonder around the map and exploit more of its features. The robot builds an internal map of the sensed points (called `knownPoints` in the code) and the points where it had been (called `beenThere` in the code), which serve as the basis of the decision making. These are based on the commanded movements and sensor data, so they are a rough estimate of the surrounding space.

The points are used to build an *Artificial potencial field* (U), and drive the robot downhill (i.e. $-\nabla U$) [1]. Since the points where the robot had been is also used for

this field (with different weight than the walls), the robot is driven towards unexplored locations. The result can be seen on Figure 5.



(a) (b)

Figure 5: Simulation with artificial potential field as guidance ($\epsilon = 0.3$). (a) Initial path. (b) Path after some time.

The effect of the potencial field can be tuned by the constant ϵ, so the commanded turning angle is:

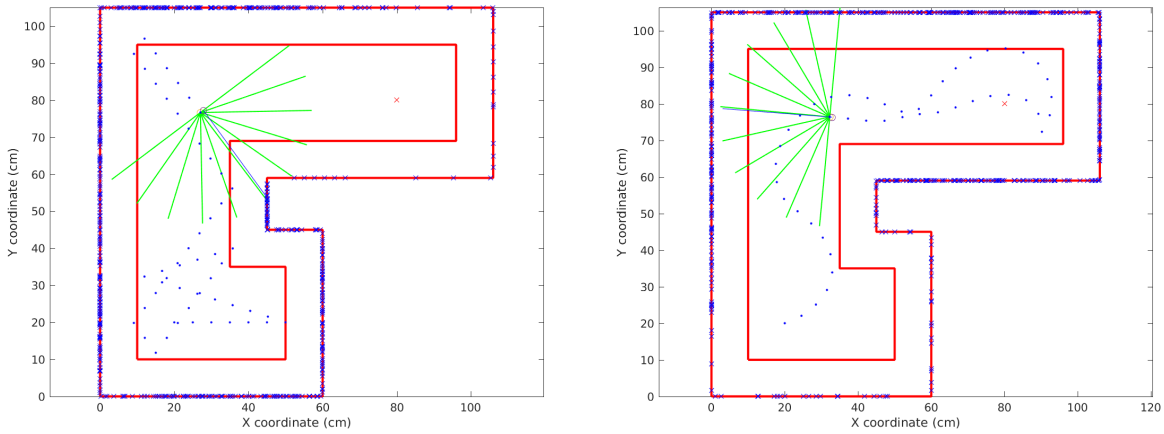
$$\delta\theta = \epsilon \cdot \arg(-\nabla U), \tag{1}$$

meaning that with lower ϵ the robot has higher inertia (goes more uphill). The commanded forward movement is kept constant, so the robot gives more consistent response even on higher gradient values (e.g. close to a wall).

This feature improves the robot's movement compared to the simple bouncing, but it needs the collision avoidance as a simple backup. The result is shown on Figure 6b.

3.2.3 Path planning

After the PFL is converged, the robot can plan its path from its current location to the target. The planning algorithm was a standard A* search [2] on a visibility graph [1] defined by the map, robot and target. This way the search space for the optimal path is greatly reduced and the movements are smoother compared to a grid-word representation. Figure 7 shows the visibility graph with the optimal path between the robot and the target.



(a) (b)

Figure 6: (a) 'Wall bouncing', with a distance threshold of 10cm (inner boundary). (b) Combined exploration (artificial potential field with $\epsilon = 0.2$ and collision avoidance), the former explores the space, a latter makes sure that the robot does not collide with the wall in any circumstances.

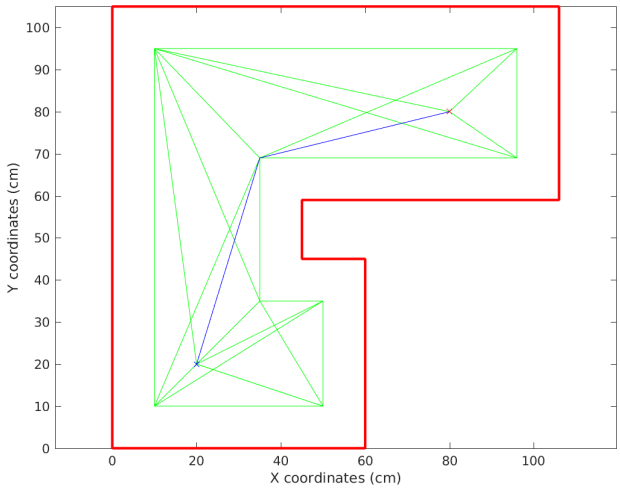


Figure 7: A* search on the visibility graph finds the shortest path (blue line) between the robot and the target (blue and red cross respectively). The modified map is used, so the robot does not go closer to the walls than 10cm.

4 Results

4.1 Simulation

Map	Avg. completition time(s)	Avg. distance from target(cm)	collision(%)
1	5.33	0.67	0
2	8.01	0.57	0
3	14.74	1.03	0

Table 1: Results without noise

Map	Avg. completition time(s)	Avg. distance from target(cm)	collision(%)
1	2.81	0.25	0
2	3.63	0.35	0
3	10.08	5.17	0

Table 2: Results with noise on sensor (±2 cm), motion (±1 cm), turning (±0.1 rad)

Table 1 and 2 show the results obtained after running the simulation 20 times each, with and without noise. As seen, the average competition time is improved in the second table adding noise, which helps the robot to reduce the time and distance from the target.

In order to avoid collisions an important feature to set is the distance to recreate the map, these tests where conducted with a distance of 10 cm, other tests where conducted with a distance less than 5 having a collision percentage lower than 2%. Later in the discussion section we will talk about the variable setting differences between the simulation and the real robot.

4.2 Experiments

To test the performance of the robot it was tested in a physical setup. On top of the modelled noise there was also considerable false ultrasonic sensor readings due to the low angle between the sensor ray and the walls, which made the navigation harder, and also the time required to make the movement made the whole process slower.

The time it takes to get to a random target point on the map is typically between 60-200s depending on the starting point. A simple signal conditioning

5 Discussion

References

[1] Howie M. Choset, ed. *Principles of robot motion: theory, algorithms, and implementation*. Intelligent robotics and autonomous agents. Cambridge, Mass: MIT Press, 2005. ISBN: 978-0-262-03327-5.

[2] *Introduction to A**. http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html. (Visited on 04/16/2016).