Implementation of a bearing-only localization simulator via particle filter

Alberto Dallolio, ID 1719939

I. Background Implementation

The project has been developed is Matlab environment with the support of the already present built-in functions. It consists on the coded implementation and development of a simulator, that allows an user to simulate the motions of a *differential drive* robot, and a particle filter capable to estimate the robot position in a map thanks to landmarks.
This work has been organized so that the fundamental components are encrypted in *classes*. Each class presents *methods* and *properties*, whose accessibility can be suitably set and modified. This organization is not hierarchical, indeed each class is a subclass of the *handle* and not of any other class. Nested classes we're not needed and their management in terms of implementation resulted to be easier thanks to their equal level of importance.

The first implemented class (and therefore the main one) is the class *Robot*. Indeed it contains all the properties and the methods that describe the features and all the possible behaviors the vehicle can adopt. In its properties we find dimensions, speed and steering limits, encoders noise and all the specifications initially empty but devoted to describe the robot in later uses. The methods consist of all those needed to *make the robot do something*. As in any class there's the object constructor, creating the robot object (in this case). Other methods were suitably coded to initialize the robot, start the simulations, compute the robot odometry, return encoder values and of course plot the robot.

The main idea is to create envelopes for every object needed later. This thought led me to develop a class for the map environment in which the robot moves, one for the path the robot may go through, one for the bearing sensor and one for the filter.

II. Robot Motion Model

I start introducing the robot and its features. The robot is a differential drive vehicle. Differential drive means that the rear wheels are independently actuated: two separate motors drive the wheels placed on either side of the robot body. Thus, it can change direction by varying the rate of rotation of the actuated wheels. This means that no other actuators are needed to control the steering: if the vehicle is a tricycle, the front wheel can simply be a caster wheel. The choice of a differential drive robot, widely employed in robotics applications, comes from its easiness to be controlled and programmed. The dimension in which the robot lives is the planar dimension on the XY plane. Indeed, the only variables needed to describe it are $[X, Y, \theta]$, the coordinates of a representative point (the robot centre) and the heading angle (angle between the pointing direction of the robot front tip and the x axis). These are the only variable that allow us to describe the robot motion. Localize the robot, means building a function that makes the robot able to guess

correctly the values of those three variables (the aim of this project).
The odometry values are computed through *Euler integration*:

$x_{k+1} = x_k + v_k * T_s * cos(\theta_k)$

$y_{k+1} = y_k + v_k * T_s * sin(\theta_k)$

$\theta_{k+1} = \theta_k + \omega_k * T_s$

where $T_s = t_{k+1} - t_k$ is the sampling time.

Thanks to these integration the robot odometry can be obtained, a vector of two elements (robot position and heading angle) that changes depending on the controls applied to the robot.
Depending on the kind of simulation the way the controls for moving the robot are invoked changes. To this purpose, I have implemented a function that asks the driver object which are the proper controls (speed and steer) needed depending on the target to be fulfilled. If the robot moves randomly in the map, a stream of pseudo-random numbers is generated and the vehicle moves through each point and the control function would ask the driver to compute the correct controls to reach each waypoint of the trajectory. Once the controls are obtained, they are passed to the *update* function that updates the robot state and its odometry. All the history of the vehicle's states are collected in a suitably defined vector, needed for plotting purposes.

I. The Driver

The driver is an object capable to suitably drive the robot. The main reasons for implementing comes because there has to be a systematic procedure that depending on the target path of the robot computes the commands to succeed. The driver object represents effectively a driver for the vehicle. It "knows" the dimensions in which the requested path has to bounded and the speed (1 m/s if not specified). The driver object is strictly needed when the robot runs randomly in the map. I have defined a pseudo-random series of numbers coming from a stream. The stream I chose is the Multiplicative Lagged Fibonacci. The random path generation is performed by selecting randomly points from the previously generated stream, simply. Once the goal is set the commands (speed and steer) are computed and passed back to the robot. The speed is simply the driver speed whereas the steering angle is computed with atan2 function with the chosen goal and the robot current position. An important thing to notice is that among the driver properties there's a variable indicating the proximity between the robot current position and the current goal towards which it is heading. This proximity value is set once the driver is called for the first time. Once the distance between the robot and the current point falls inside this proximity, the function setting the successive goal is invoked. However, as soon as the robot turns its direction towards the next goal waypoint, the distance from the previous is set to infinity, so that it won't be chosen anymore. So doing, the simulation runs faster and fluently without interruptions.
All this explanation has not to be considered for the other two kinds of simulations: the one in *chosen mode* and the one in which the robot commands are specified by the user.

II. The Chosen Modality

Chosen Mode refers to the modality thanks to whom the user can directly specify the points the has to go through. It seems obvious that the functions needed for updating the robot state, computing the controls and odometries are the same, except for slight modifications. Those modifications regards the fact that the waypoints are not chosen randomly anymore, but are specified by the user manually and graphically on the map. Indeed, *ginput* Matlab function allows this. Therefore, once the points are selected on the map the simulation starts and the driver is again called. It's duty is that of computing again the control commands to drive the robot among the selected points. Once the robot reaches the first points the commands for reaching the next are already computed and it adjusts its heading, and so on till it stops exactly on the last point.

III. Map, Landmarks and Sensor

I. Map and Landmarks

The map object is defined thanks to a suitably defined class called Map. The map consists of a simple squared cartesian map. The user can specify both the map dimensions (XY axes length) and the number of landmarks. The landmark population is a collection of fully colored circles with fixed dimension. I developed a short algorithm able to assign a random color to every landmark each time the map object is generated. The possible colors for a landmark are red, blue and cyan.

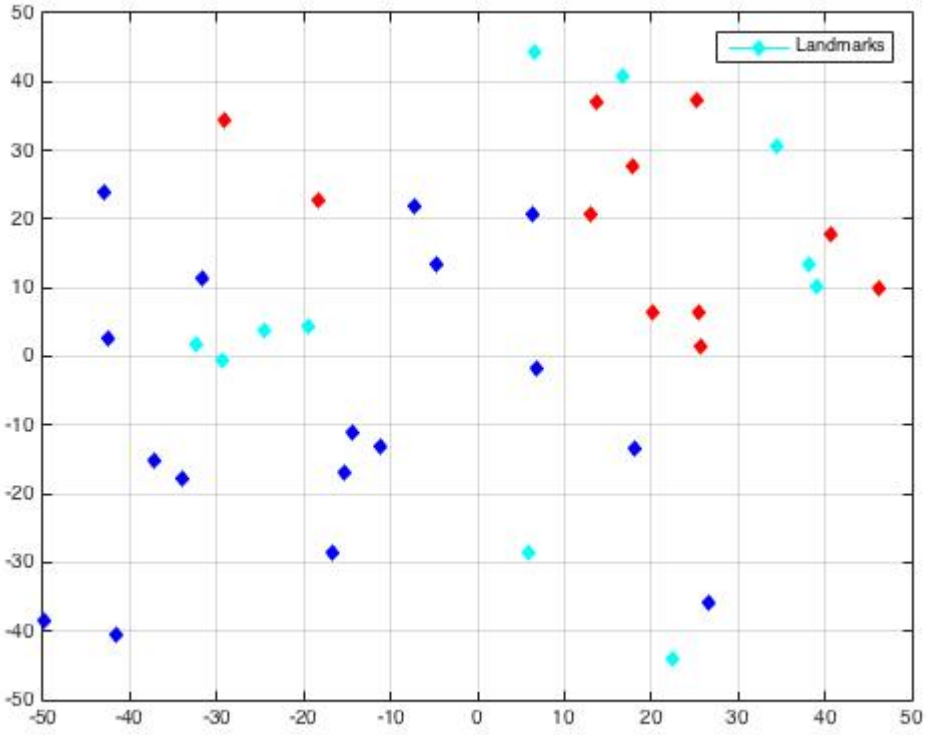


Figure 1: Map with dimensions [-50 +50] and 40 randomly colored landmarks.

II. The Sensor

The coordinates of the landmarks is one of the main variable of the system. Indeed on those values depend the measurement readings of the sensor. The sensor is a bearing-only sensor capable to detect the bearing angle between itself and any landmark in the map. Talking about the implementation, it has been coded into a class called *BearingSensor*. This kind of sensor is able to measure both the bearing to a landmark and its class (color). The sensor object creation sets the fundamental features of the sensor, like the map and robot objects acquisition and the limit angles that define the robot field of view in each time instant. This means that if this boundary is set to 30°, the sensor would intercept only those landmarks falling within a cone with angle 30°spacing ahead the vehicle. The sensor properties are the noise (a white gaussian noise) capable to affect both the bearing measurement and the the class evaluation. Since I always impose the a range window of [-180 +180] the sensor can see all the landmarks simultaneously. Indeed it can both compute the bearings to all the landmarks and also compute the bearing to a randomly selected landmark. The latter option is employed by the filter for the localization. The filter, in fact, uses the bearing angle of one single reading to be compared to its corresponding estimation.

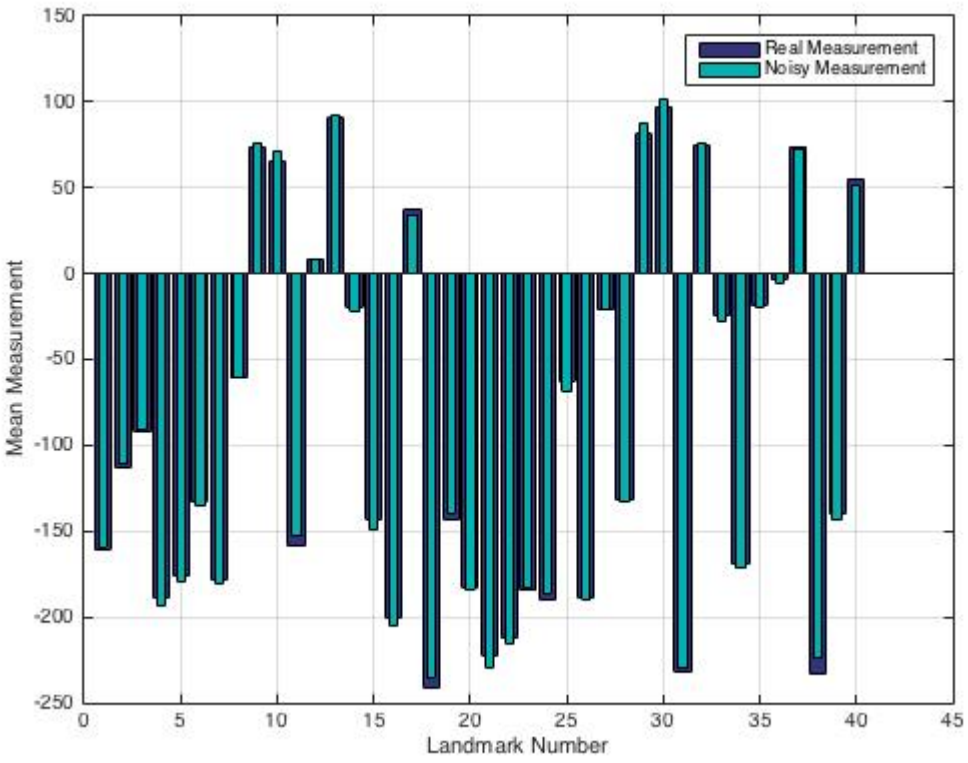


Figure 2: Histogram of the sensor measurements of 40 landmarks

IV. A brief introduction: Bayesian Filtering

Bayes Filtering is the general term used to discuss the method of using a predict/update cycle to estimate the state of a dynamical system from sensor measurements. Generally, the Bayes Filter Algorithm computes the *belief* distribution (*bel*) on the base of measurements and general control data. It works in a recursive way so that the belief $bel(x_t)$ at time t is computed from the belief $bel(x_{t-1})$ at time $t-1$. Indeed, the inputs are the previous belief $bel(x_{t-1})$ together with the current control u_t and measurement z_t. Of course, the output is the belief at time t, $bel(x_t)$. As previously stated, a Bayes Filter Algorithm operates cyclically: the prediction phase comes before an update (or *correction*) one. The latter, is responsible for the calculation of the current belief on the base of the previous one.

Algorithm 1 The Bayes Filter Algorithm.

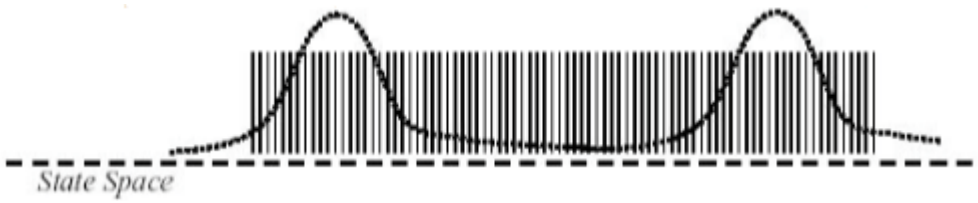
BayesFilter[$bel(x_{t-1})$, u_t, z_t]:

1: **for** all x_t **do**
2: $\quad \overline{bel}(x_t) = \int p(x_t|u_t, x_{t-1}) bel(x_{t-1}) dx$
3: $\quad bel(x_t) = \eta p(z_t|x_t) \overline{bel}(x_t)$
4: **end for**
5: **return** $bel(x_t)$

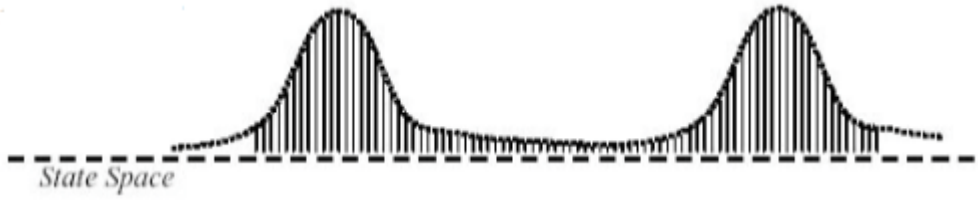
In the first step the control u_t is processed. Hence, the belief of x_t based on the prior belief and the control is multiplied by the belief of the previous state. $\overline{bel}(x_t)$ is represented as an integral (sum) of the product of two distributions: the prior assigned to x_{t-1} and the probability that u_t induces a transition from x_{t-1} to x_t.
The second step is the measurement update, in which the previously obtained belief $\overline{bel}(x_t)$ is multiplied times the probability that the measurement u_t may have been observed. Since the result is not a probability (an integral action would be useless), its value is normalized (η), leading to the final belief.
Some considerations can be carried out. First, it's worth to outline the importance of the two probabilistic laws: the state transition distribution and the measurement distribution. The former describes how the state changes over time, whereas the latter characterizes how many measurements are governed by states. Second, this algorithm makes a *Markov assumption*: the state is a complete summary of the past. Thanks to this, it is possible to infer that the belief of the state is enough to represent the past history of the system.

V. The Particle Filter

The basic idea of particle filters is that any probability density function can be represented as a set of samples (particles). Each particle has one set of values for the state variables. This method can represent any arbitrary distribution, making it good for non-Gaussian, multi-modal probability density functions. The key idea is to find an approximate representation of a complex model (any arbitrary probability density function) rather than an exact representation of a simplified model (Gaussians).



Call u the input, z the observation and d (data) both inputs and observations. The problem I face now is hot to find the samples (or particles). Basically, what I want to sample is the posterior: $p(x_t|d_{0..t})$. But since I do not have an explicit representation of it I can just rely on sampling my prior belief. Indeed, from the previous time step I have a belief that I know how to update thanks to my kinematic motion model. Thus, sampling my prior belief I can update each sample on the basis of the observation made and the prior belief itself, making use of the so called "importance weights". This is a sort of comparison between the posterior and the prior belief.
The first thing to do is sampling from your prior (any kind of distribution). For each of those samples, I can find the value of the posterior (let's call it p(x)).



So for each sample, I assign that sample a weight, w(x), equal to p(x)/q(x), where q(x) is the prior belief. At this point, when the particles are weighted, I can use my highest-weighted (corresponding to highest-probability) sample as the "best guess" state. To represent the probability properly with samples, though, I need the density of the particles in any segment of the state space to be proportional to the probability of that segment. So in order to adjust the densities properly, I resample the particles. In doing this, I keep the total number of samples the same while changing their distribution: more particles in the segment of state space in which the probability is higher and a decreasing number of particles in low-probability regions. To this purpose, particles are re-distributed on the basis of importance weights. Note that the particles with high probability can be chosen multiple times, while those with a low probability can not be chosen at all. The picture below shows graphically what just explained.

VI. The Importance of Resampling

One question that might easily arise is: why resampling is necessary? The answer is that if I just keep my old particles forever without resampling them, what happens is that they will drift around according to the motion model (transition probabilities for the next time step), but other than their weights, they are unaffected by the observations. What may happen is that the unlikely particles would be kept around and pushed to even more unlikely states, leading the system to have let's say, one particle in an area of high probability of the posterior and a lot of particles in areas with almost null probability. This is of course an extreme case of what may happen. This phenomenon is called *particle depletion*, whose result is that I won't represent the probability density function properly if I don't have lots of particles in the areas with highest probability. That's why I need to keep resampling the particles at each iteration.

I. The Particle Filter Algorithm

To start the algorithm, the belief of the initial state x_0 has to be found (named $p(x_0)$), corresponding to the initial guess of the probability density function. A common solution adopted for robot localization purposes is that of scattering particles all over the map. Then, the actual algorithm loops among three sequential phases: prediction, update (or correction) and resample.
All the steps can be formalized in one unique equation, whose components are analyzed in the following equation:

$$p(x_t|d_{0..t}) = \eta p(z_t|x_t) \int p(x_t|x_{t-1}, u_{t-1}) p(x_{t-1}|d_{0..t-1}) \mathrm{d}x_{t-1} \tag{1}$$

where the integrated part is the prediction, the factor before the integral represents the update and the result is the resample. It is derived by applying Bayes rule to the posterior, and then using the Markov assumption. The algorithm states that I can end up in the same state in time t from more than one state in time *t-1*, and thus integration is needed. Actually I won't care about this since the particle representation already takes care in the integration. During the algorithm execution, each component is computed and involved in this final equation. First I start with the probability density function from the last time step, that is multiplied by the motion model in the "prediction" step to get q(x), the prior probability. Next, importance weights w(x) are computed using the perceptual model and normalize them so that they sum to 1. Finally, the posterior probability (our belief about the state variables) are computed as q(x) times w(x).

Algorithm 2 The Particle Filter Algorithm.

ParticleFilter[X_{t-1}, u_t, z_t]:

1: $\overline{X_t} = X_t = 0$
2: **for** $m = 1 to M$ **do**
3: sample $x_t^{[m]} \sim p(x_t|u_t, x_{t-1}^{[m]}))$
4: $\omega_t^{[m]} = p(z_t|x_t^{[m]})$
5: $\overline{X_t} = \overline{X_t} + \langle x_t^{[m]}, \omega_t^{[m]} \rangle$
6: **end for**
7: **for** $m = 1 to M do$ **do**
8: draw i with probability $\propto \omega_t^{[i]}$
9: add $x_t^{[i]} to X_t$
10: **end for**
11: **return** X_t

At line 3 the algorithm generates a hypothetical state x_t^m for time t based on the particle x_{t-1}^m and the control u_t. The index m indicates that it is generated by the m-th particle in X_{t-1}.
Line 4 calculates for each particle the *importance factor*, ω_t^m, used to incorporate the measurement z_t into the particle set. If we interpret ω_t^m as the weight of a particle, the set of weighted represents (approximating) the Bayes filter posterior distribution $bel(x_t)$. By incorporating the importance weights into the resampling process, the distribution of the particles change. Indeed, the resulting sample set usually contains many duplicates, since particles are drawn with replacement. More important are the particles that are not contained in X_t: those are the particles with lower importance weights.

The resampling step has the important function to force particles back to the posterior $bel(x_t)$. The idea comes from Darwin: survival for the fittest, since it refocuses the particle set to regions of the state space with higher posterior probability.

It can be noticed that the particle filter approximates the posterior by a finite number of parameters. Here, the key idea is to represent the posterior, before called $bel(x_t)$ by a set of random samples drawn from this posterior. The samples of a posterior distribution are called *particles*: each particle is a instantiation of the state at time t, that is, the hypotheses of how the true world state may be at time t. Hence, the intuition is to approximate the belief $bel(x_t)$ by the set of particles X_t. The denser a region of the state space is populated by samples, the more likely it is that the true state falls into that region. The particle filter algorithm constructs the belief $bel(x_t)$ recursively from the belief $bel(x_{t-1})$ and since beliefs are represented by sets of particles, this means that the particle set X_t is built recursively from X_{t-1}.

I.1 Prediction

In the prediction step, to each particle, a random sample from the motion model is added. Since each particle represents a possible state of our system (the robot in our case), what happens is that when the robot starts moving, its the ending position will be one among a cloud of particles (states). This means that the resulting distribution of particles approximates the prior distribution.

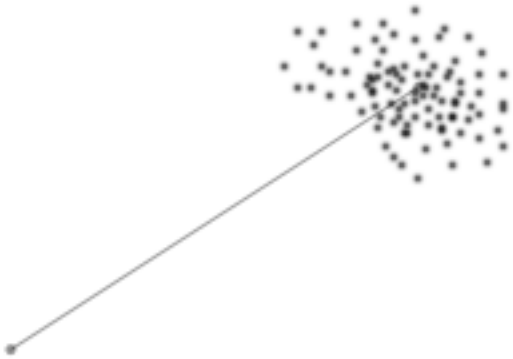


Figure 3: The resulting distribution approximates the prior distribution

Hence in the prediction step, the prior probability is obtained, computed as the multiplication of the probability density function from the last time step times the motion model.

I.2 Update

In the update step, sensor measurements are taken and each particle is assigned a weight that is equal to the probability of observing the sensor measurements from that particle's state. The weights are normalized so that their sum is again 1. This leads to have particles with higher weights with respect to others. The weight associated to each particle is computed as:

$$w(x) = p(x)/q(x) = p(z_t|x_t) \tag{2}$$

I.3 Resampling

The idea is that a new set of particles is chosen such that each particle survives proportionally to its weight. Thanks to the resampling phase, the weighted cloud of particles turns into somewhat more condensed and smoother cloud of unweighted particles. Unlikely, particles at the fringe of the cloud will be discarded as well as the particles closer to the center of the cloud will be replicated so that the region with high probability has a consistent density. So doing, the process is able to represent properly p(x), the posterior distribution.

Particle filters, on the other hand, can keep track of as many hypotheses as there are particles, so if new information shows up that causes you to shift your best hypothesis completely, it is easy to manage.

VII. Particle Filter Implementation

The particle filter has been implemented as usual exploiting methods and properties of the specific filter class. Its main features are: the number of particles, the weights to be assigned to each particle, the likelihood and model noises, the standard deviation, the estimated states and the particle states.
After the filter object has been created, the initialization occurs. The initialization consists on creation of a set of particles to be scattered on the map. The particles are initially randomly distributed. During the initialization, all the weights are set to unity.
Running the filter means triggering the cycle of operation explained before: prediction, update and resample. Indeed, once the filter run has been triggered the cycle iterates for a number of times chosen by the user. Initially, the robot moves one step and the odometry is computed. The odometry is needed for the prediction step, in which thanks to the odometry the noisy next state of each particle is computed. Then the sensor provides the measurement from one randomly chosen landmark. This is needed because the update consists of computing the bearing from all the particles to that landmark. Thus the innovation is computed as the real bearing measurement from the robot to the landmark subtracted to the bearing from each particle to that landmark. Hence, for each particle the innovation is computed and the respective weight is assigned. The particle filter requires a likelihood function that maps the error between expected and actual sensor observation to a weight (a 2D Gaussian is used to this purpose).
Once the weights have been assigned to each particle, the resampling process can take place. Thanks to this last step the cloud of particles turns into somewhat more condensed.

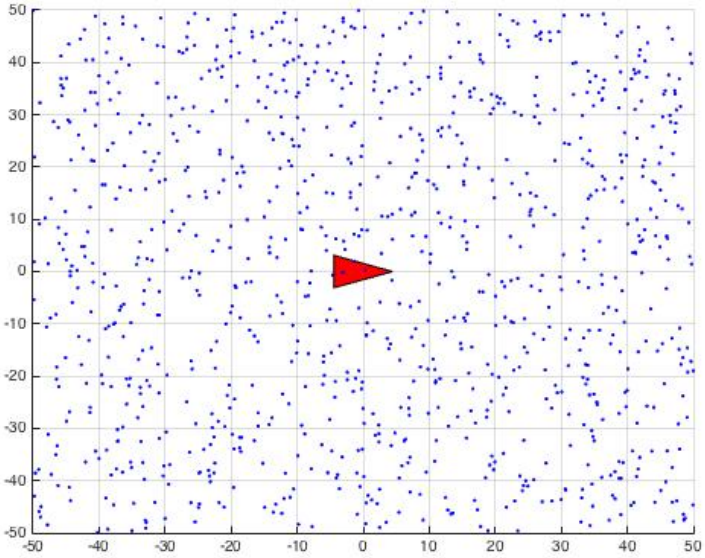


Figure 4: Filter initialization: scattering particles.

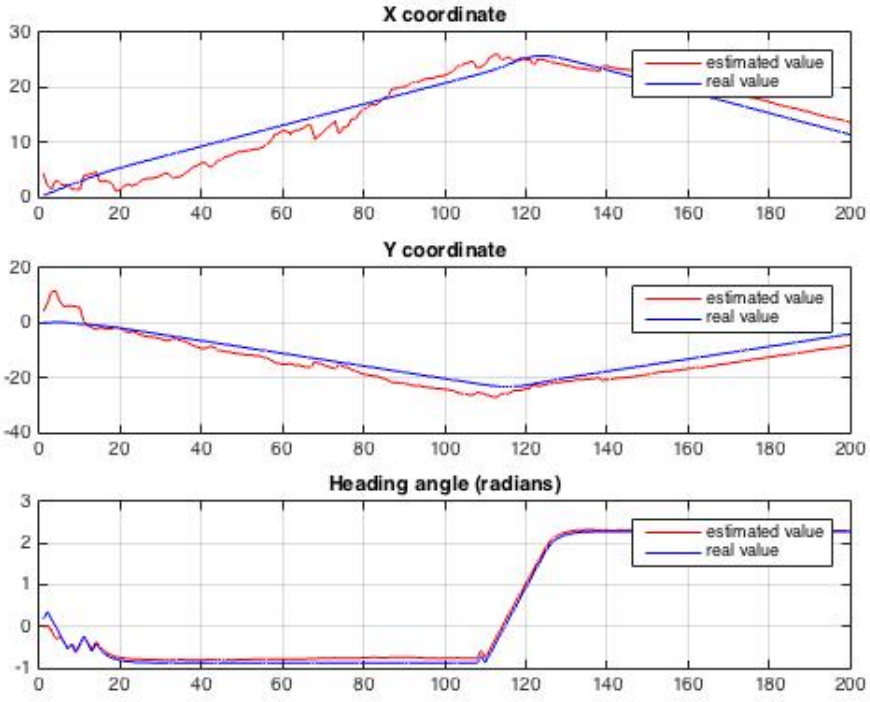


Figure 5: State profiles: real and estimated.

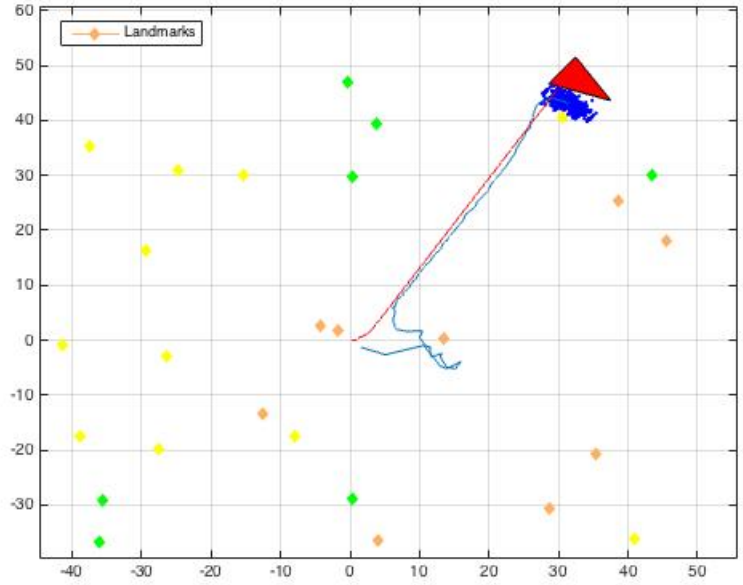


Figure 6: Robot and particles paths

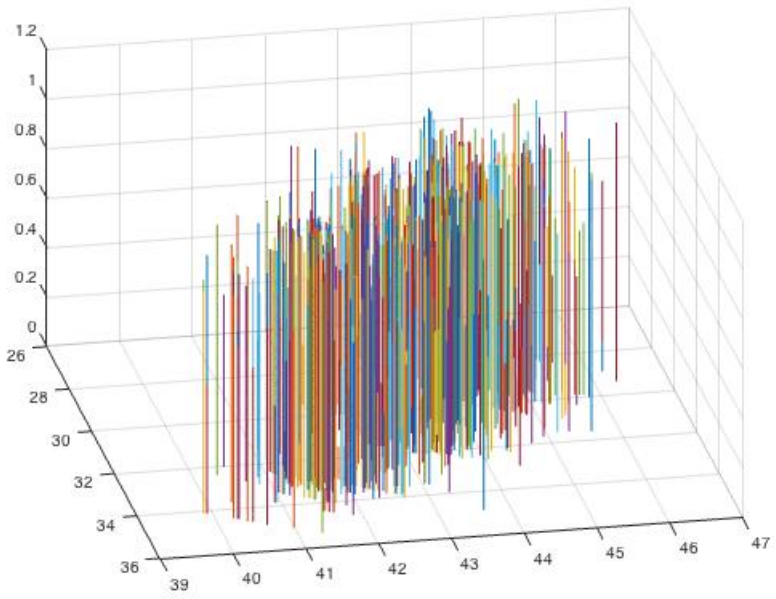


Figure 7: Pdf plot

VIII. Simulations

I. Random Simulation

The first implemented simulation is the one in which the robot runs a random path around a specified map. The only value to be specified by the user is the number of steps of the simulation. The outputs, instead, are the measured bearings at each step, the odometry at each step and the classes of the landmarks recognized by the sensor. The path executed by the robot is specified by the *randstream* generator, that in this case creates a stream of waypoints based on the Fibonacci series generator. The simulation time depends on the chosen number of time steps.

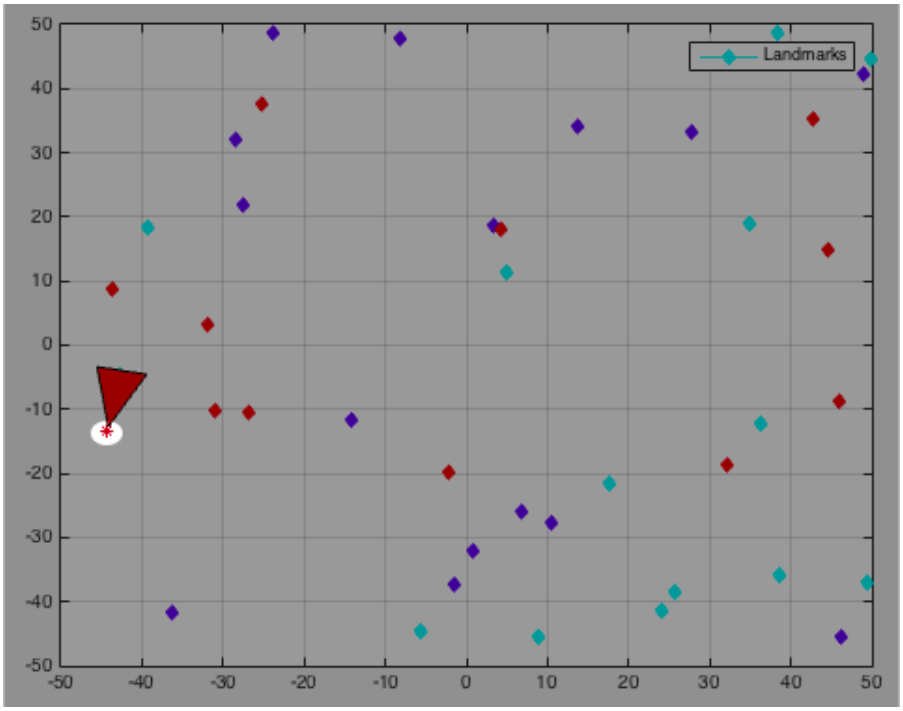


Figure 8: Vehicle approaching one of the random waypoints.

```
>> robot.run_rand(1000,map,sensor)
goal: (12.1 -19.3)
goal: (-15.7 -40.8)
goal: (-13.7 9.0)
goal: (-47.9 37.0)
goal: (-20.6 49.3)
goal: (41.2 10.1)
goal: (-12.0 30.4)
goal: (-25.3 37.8)
goal: (14.7 -39.4)
goal: (14.9 -29.6)
goal: (-26.6 -47.1)
goal: (-7.1 19.9)
goal: (4.9 0.4)
goal: (34.7 43.3)
goal: (-4.1 -30.8)
goal: (7.6 -9.6)
goal: (9.6 -31.9)
goal: (15.8 18.2)
goal: (-45.0 -42.4)
goal: (-32.5 -34.0)
goal: (27.9 -48.8)
goal: (44.2 -24.1)
goal: (7.7 -23.2)
goal: (-43.8 -11.4)
goal: (-47.9 -38.2)
goal: (18.1 20.0)
goal: (45.1 2.1)
goal: (27.2 -39.7)
goal: (-13.9 1.3)
goal: (5.2 38.3)
goal: (-20.4 32.5)
```

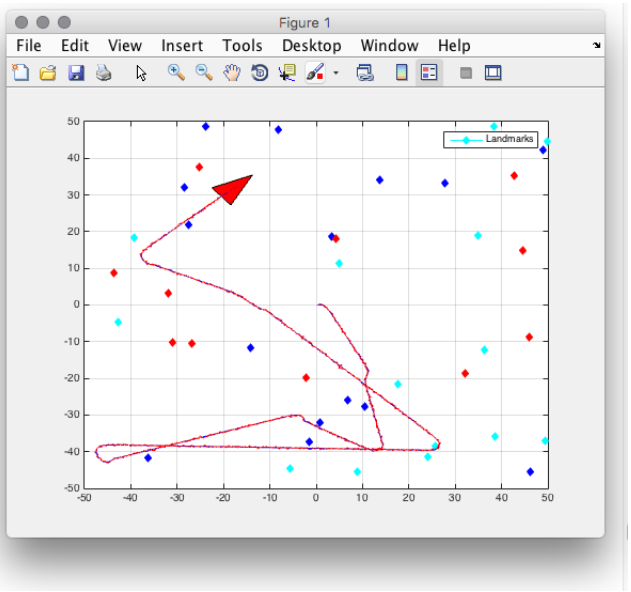


Figure 9: Simulation running and goals generation.

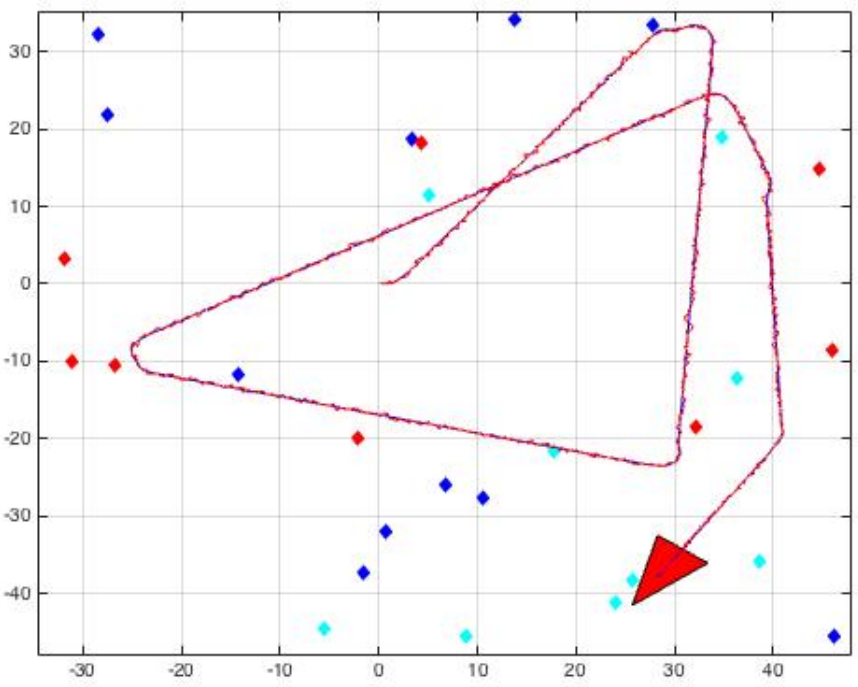


Figure 10: Robot real and noisy path.

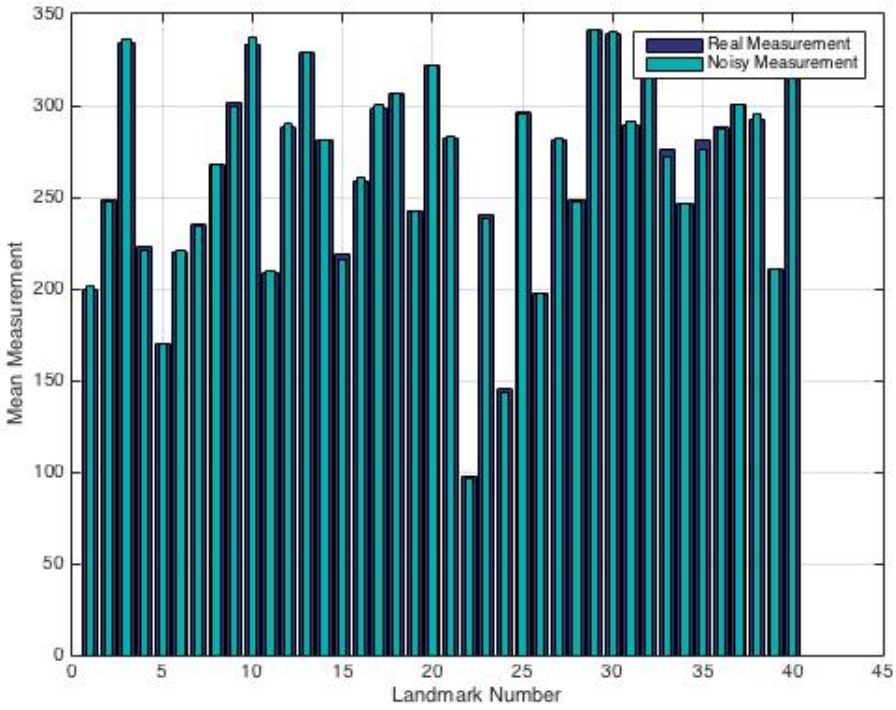


Figure 11: Bearing sensor measurements.

II. Chosen Mode Simulation

As explained before in this simulation modality the user can input manually and graphically the waypoints on the map. He can choose freely the number of points (also one thousand) and their location in the planar space.

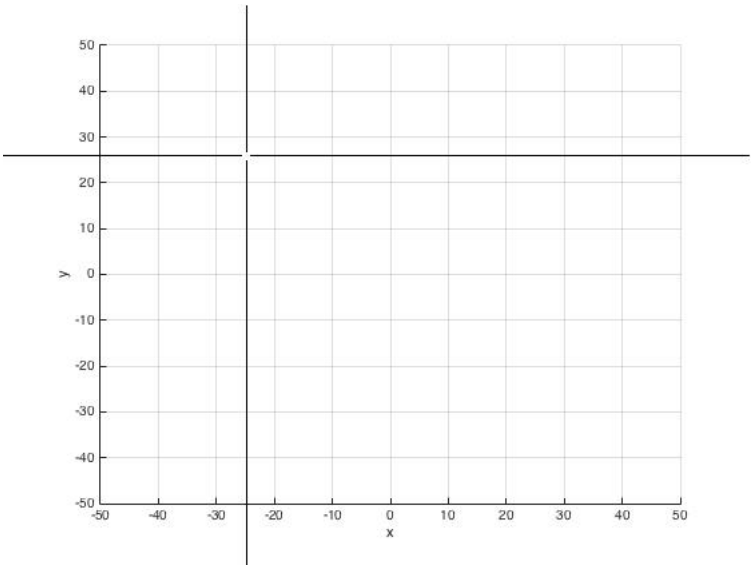


Figure 12: Choosing the points with *ginput*.

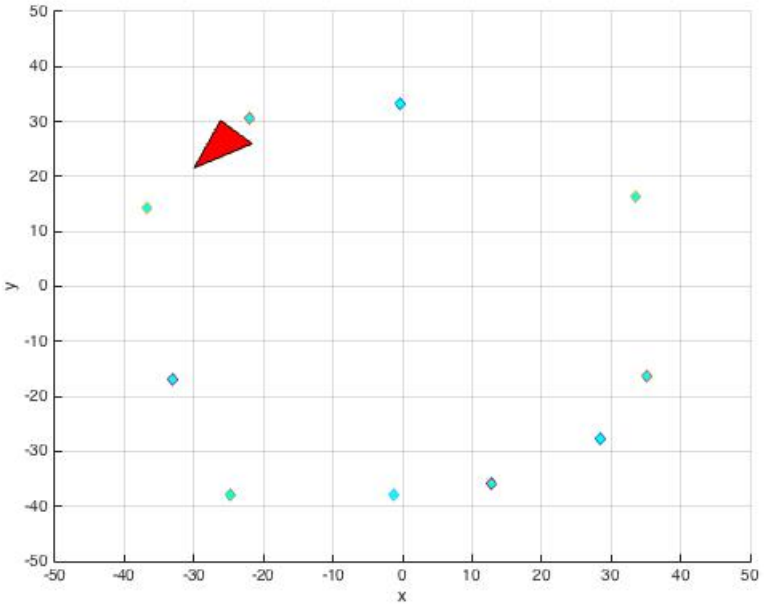


Figure 13: The robot runs from one to another in the same order of choice.

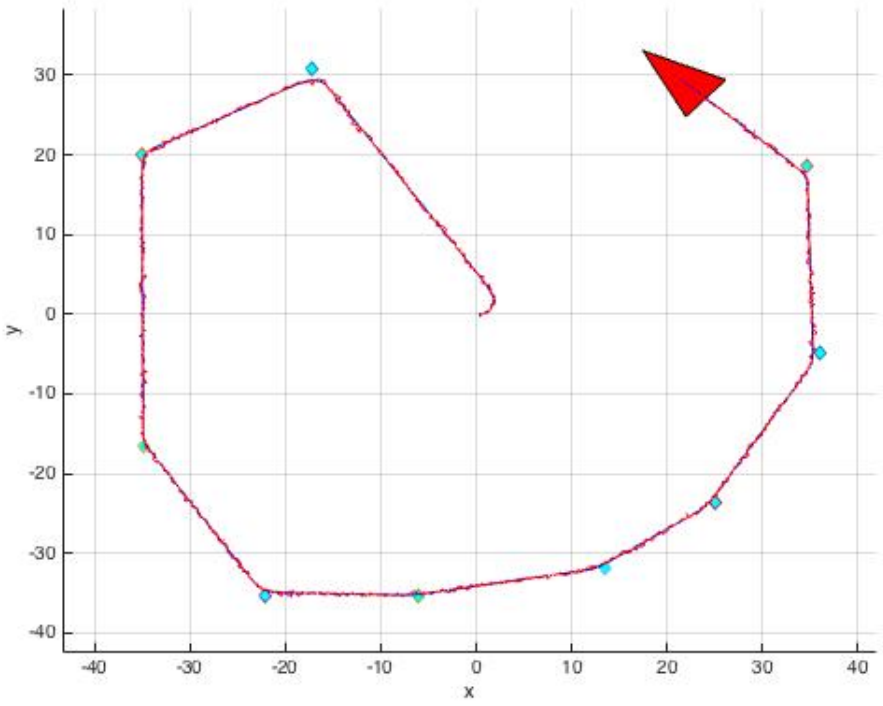


Figure 14: Real and noisy path plot.

The robot runs from the first to the last point in the same order the user chose them. Finally, the odometry is given as output as well as the noisy and real path executed by the robot.

III. Specific Simulation

The last kind of simulation that I implemented gives the user the possibility to specify precisely the steering angle, the speed and the time the simulation has to last. This allows the generation of specific trajectories depending on the given commands.

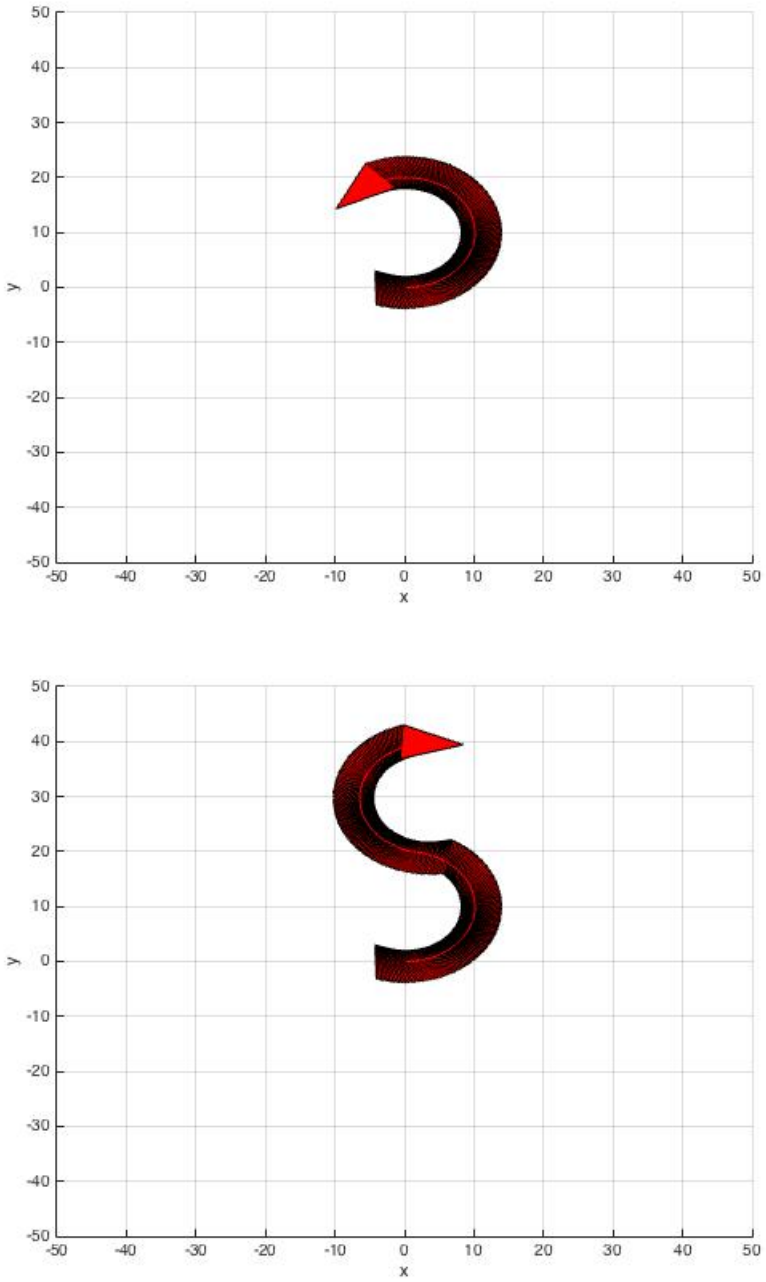


Figure 15: Precise instructions are given to the vehicle.

In the first snapshot one single speed and steering angle were given to the robot, indeed running a circle. Whereas in the second two different values for speed and steering angle were given, together with a clip value responsible for the change of the vehicle behavior. Again, all the robot states are given as output.

References

[1] Sebastian Thrun Wolfram Burgard Dieter Fox, *Probabilistic Robotics(Intelligent Robotics and Autonomous Agents)*, The MIT Press ©2005

[2] Siciliano, B. Sciavicco, L. Villani, L. Oriolo, G. *Robotics: Modelling, Planning and Control*, Springer-Verlag 2000