# RBPF-SLAM in *MatLab* Users Guide

Adrian Llopart Maurin (P.h.D.)

Kongens Lyngby 2016

Technical University of Denmark
Department of Electrical Engineering
Ørsteds Plads, building 326,
2800 Kongens Lyngby, Denmark
Phone +45 50285623
adllo@elektro.dtu.dk
https://github.com/Allopart

# Abstract

Rao-Blackwellized Particle filters have been introduced in simultaneous localization and mapping (SLAM) effectively in the past years. Improved techniques have helped reduce the number of particles needed and allowed a faster and more robust solution. These solutions can be seen, and used, in the C++ ROS *gmapping* package. This manual presents the conversion to a simpler but easy-to-understand *MatLab* implementation and will describe the RBPF-SLAM principle in a similar manner.

# Contents

Chapter 1

# Introduction

One of the clearest functionalities of mobile robots is building maps of its surroundings and navigating through them; this is often referred to as SLAM, Simultaneous Localization and Mapping. There exists, however, certain difficulties when implementing SLAM, mainly, for a robot to localize (know its true position), a very precise map has to be built before; but, for a very precise map to be built, a robot must know exactly where it is. Friction, control loss or small obstacles are often the cause of a bad odometry which leads to a poor estimate of the true position. The Rao-Blackwellized Particle filter (as shown in [1] and [2]) solves this issue by generating estimates of the possible position (particles) and giving them a weight. Those particles that have a higher weight, because their estimate matches better the reality, will survive, and the rest will die out in the next generation. To keep a continuous amount of particles and not let all of them die out, resampling stages are carried out where those surviving particles reproduce and keep the particle count constant. It is evident that the more particles used, the more possible descriptions of the reality one has and the higher probability of having at least one particle which is almost perfect. This also induces a higher computational necessity and stops the solution from becoming a real-time application.
The real issue is, therefore, weighing these particles to know how close they are to the reality. This is known as the *proposal distribution* $\pi$, and in the latter iterations of the RBPF-SLAM solution, optimal techniques have been developed to obtain a more accurate representation, whilst keeping the particle count to a

minimum.
In summary, what the RBPF-SLAM algorithm does is given an initial estimate of the true pose, it will use the high precision of a laser scan and the latest map generation to converge the estimate to the true position. This leads to a more accurate map representation and a considerable reduction in errors and number of particles.

## 1.1 MapGeneration

### 1.1.1 Introduction

In a real-world application of the RBPF-SLAM, the real odometry and laser scans of a robot would be used. For a simulation, however, these values need to be generated by the user. To do so, a simulation map will be created, and a simulated robot will move through it following a determined path. Laser scan data will be derived mathematically and will be fed, together with a error-incorporated odometry, into the RBPF-SLAM method, that will generate different particles and maps to better adjust to the reality of the simulation.

### 1.1.2 Creating the map

The construction of the map is done through individual walls. A matrix *map* (with dimensions 4xn) will represent these walls. The first row corresponds to the x-coordinate of the initial point of the wall; and the second row will be the y-coordinate. Similarly, the third and fourth row are the x and y coordinates of the final wall point, respectively.

$$map(:,n) = [x_{initial}; y_{initial}; x_{final}; y_{final}]; \tag{1.1}$$

Therefore the user can create whichever map he/she feels fit, as complex or simple as wanted. the result can be visualized uncommenting the next few lines of code.

### 1.1.3 Path generation

Once the map is complete, the path the robot will follow has to be detailed. This is done in the *via* vector which incorporates the coordinates of points in the path the robot has to travel through. The *MatLab* function *mstraj* by Peter Corke uses the defined *via*, maximum axis-speed limits and a timestep to determine the trajectory (*path*) of the robot.

### 1.1.4 Control parameters

This is precisely where the control aspect of robotics comes into place. A controller has to be developed to make the robot follow the path as desired. in other words, you will pre-program the robot to follow a certain path instead of giving it movement commands at every time step. The parameters that have to be tuned are $d$, $K_p$ and $K_h$. The error signals in the feedback control loop are calculated as follows:

The absolute distance from the desired position (in *path*) and the latest position $x_{t-1}(x, y, \theta)$ given by the latest movement control command $u_{t-1}$:

$$e = \sqrt{((path_{1,t} - x_{1,t-1})^2 + (path_{2,t} - x_{2,t-1})^2)} - d; \tag{1.2}$$

The angular error between these two positions:

$$th = atan2((path_{2,t} - x_{2,t-1}), (path_{1,t} - x_{1,t-1})); \tag{1.3}$$

The new movement control commands will be:

$$u_{1,t} = K_p \cdot e; \tag{1.4}$$

$$u_{2,t} = K_h \cdot (angdiff(th, x_{3,t-1})); \tag{1.5}$$

The resulting path after tunning the controller can be visualized commenting out the RBPF line (**[pf.xh(:,k),pf] = RBPF(x(:,k),y,LRS,u(:,k-1),a,pf, Nsamples,'multinomial_resampling')**) and uncommenting the true position plotting section that follows it.

## 1.2 MotionModel

### 1.2.1 Introduction

*Motion models* comprise the state transition probability $p(x_t|u_t.x_{t-1})$ to determine a new position given its previous one and the movement control command. Therefore, theprobailistic kinematic approach will be used (instead of a deterministic one) because it takes into consideration the control uncertainty due to noise or unmodelled exogeneous effects.
Since many commercial mobile robots are actuated by independent translational and rotational velocities, a *velocity motion model* will be used, as presented in [3].

### 1.2.2 Motion error parameters

This motion model relies on the robot specific motion error parameters $\alpha_1$ to $\alpha_6$. The values of these parameters have to be calculated for each robot specifically. The resulting odometry distribution will vary in great amounts because of them. Translational error is given by $\alpha_1$ and $\alpha_2$, the angular error by $\alpha_3$ and $\alpha_4$, and $\alpha_5$ and $\alpha_6$ represent the end rotation error.
The method *Motion_ Model_ Velocity_ Test* allows for a fast, simple and visually-appealing calculation of these values. Given a starting position $(x_{t-1})$, a movement command $(u_t)$ and a grid size $[x_g, y_g]$, it will generate the odometry proposal distribution as seen in the following figures:

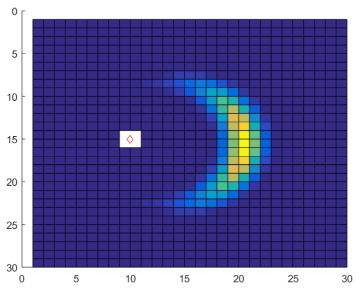

**Fig. 1.1:** Odometry distribution for a $u_t$=[10 0] and an $a$=[0.01 0.01 0.01 0.1 0.001 0.001]

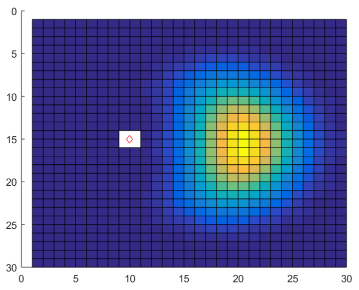

**Fig. 1.2:** Odometry distribution for a $u_t$=[10 0] and an $a$=[0.1 0.1 0.01 0.01 0.001 0.001]

It is suggested that you evaluate how big your movement commands $u_t$ are for every time step (they are dependant on the number of points *path* has) adn the *maximum velocity* set in the *mstraj* function. Once you have an estimate, use the *Motion_Model_Velocity_Test* to see how your odometry distribution would look like for different motion error parameters $\alpha$. Try to fine-tune these values so that the simulated odometry error resembles the reality.

## 1.3 RBPF-SLAM

Once the map has been correctly generated and the simulated robot follows an adequate trajectory, it is time to enable the Rao-Blackwellized Particle Filter. In the *Simulation parameters* section at the top of the *Main* code one can modify many parameters that have a direct impact on the result of the RBPF:

- *Ts*: Sampling time, by default set to happen every second.
- *n_cell*: sets how many centimetres a cell represents. The default is that each cell has an area of $10cm^2$. The modification of this value should not change the structure of the code nor give any errors, but this has not been tested out yet so it is **not guaranteed** to work.
- *NPC*: Number of particles. The more particles selected, the more accurate the occupancy grid will become, but the more computational power required for the calculations. The main advantage of the improved techniques used in this simulator for RBPF-SLAM is that the total amount of particles can be kept very low (3-6) and the results will still be fairly accurate to the reality.
- *usable_area*: When running the particle filter, only data in a radius of *usable_area* will be utilized, specifically, the occupancy map and the laser scan data.
- *sigma_v*: Describes the standard variation of the sensor beam.
- R1: represents the variance matrix for the odometry. Small changes in the variance have great effects on the filter.
- *Nsamples*: Corresponds to the number of samples to be taken when creating the Gaussian distribution.
- *max_speed*: The maximum speed of the simulated robot in the vertical and horizontal axis, taking into account the size of each cell.
- *LRS_Sensor structure parameters*: to be set according to the laser scanner you want to model.
- *L*: The occupancy grid is initialized to have a value of 0.5 in all of its cells. This will change on every iteration of the filter, updating the cell value to describe the probability of an exiting wall in that location.

### 1.3.1 Filter initialisation

The first time step of the simulation will be an initialisation of the occupancy grid map. Since we have not moved yet, the position estimate corresponds to the true value. Therefore, the generated map at this point will be very similar to the reality (except for minor laser scan variances). For this reason, the map is updated with the first scan and the particle weights and locations (which in both cases are the same for all particles) are initialised.

### 1.3.2 RBPF simulation

From the second iteration onwards, the particle filter will work continuously updating positions of particles and their maps. At the beginning of every iteration the new movement commands $u_t$ are calculated (based off the $e$ and *theta* errors mentioned in section 1.1.4). The new true position $x_t$ is calculated assuming no errors (ideal case that does not match the reality) and 'true' scan data is obtained from the simulation.
The **RBPF** function is then triggered, taking all the above as arguments. After various variable initializations, the filter starts working for every particle:
An initial estimate of the position $x_t$ is obtained from the previous particle positions $\ddot{x}_{t-1}$, the movement command $u_t$ and the variance matrix $R1$ (another option would be to sample one time directly from the motion model). An estimated laser scan is obtained from teh estimated position and the last iteration of the occupancy grid map for that particle via the **Measurement_Estimate_From_Grid** method. Before the ICP (scan-matching) algorithm can be applied, the 'true'scan $y$ and the estimated one $y_h$ have to be pre-processed. Since the laser data is in radial form (laser angle and distance) it will be easier to match one-to-one scan. To do so, all scans that have a distance bigger than *usable_ area* will be depreciated. This helps computational-wise and focuses only on scans that come from areas of the map that are closer to the robot and, thus, are more reliable. Theoretically, those scans that are neglected in $y$ should also be in $y_h$, and vice versa. In reality, sometimes they do not and those scans that mismatch because of this will also be neglected to avoid errors in the ICP algorithm. Finally, the scans are converted to Cartesian space and put in the World frame and the initial estimate position $x_t$ is also included in the point cloud for reasons described below.
The ICP algorithm is run between both datasets ($y$ and $y_h$). If the ICP reports a failure or generates translational or rotational matrices which are erroneous, the final position estimate of the particle $\ddot{x}_t$ will be equal to the initial estimate $x_t$ that simply came from the odometry distribution. In the cases where the ICP succeeds, a translational $TT$ and rotational $TR$ matrix are returned;

these represent the movement the estimated laser scan data $y_h$ has to follow to converge to the 'true' scan data $y$.

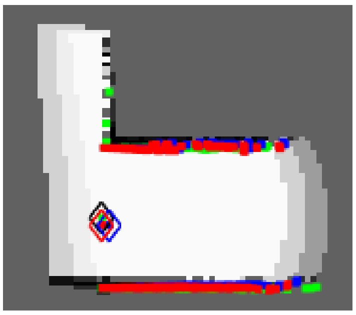

**Fig. 1.3:** Different position estimates and scans: true, before ICP and after

The trick here is that the ICP also returns the rotated data (which should, in most cases, have converged), and since the initial estimated position was included in the dataset, a better estimate of the position $\dot{x}_t$ thanks to scan-matching is already derived (the $\theta$ value will remain the same as the one previously estimated). This is automatically visualized through a figure which has, in the background, a grey-scaled image of the occupancy grid map up until that point, the diamond shaped points that represent the positions (black: true position, blue: initial estimate, red: new estimated position after scan-matcher). In the same manner, the laser scans are represented where green is the true scan, blue is the scan obtained from the previous occupancy map and an initial estimate, and red would be the converged scan.

The next step is obtaining samples $x_j$ to create the Gaussian distribution. These samples are randomly selected around $\dot{x}_t$ and will be used to evaluate the target proposal consisting of the odometry distribution $p(x_j|x_{t-1}^{(i)}, u_t)$ and the observation likelihood $p(z_t|x_j, m^{(i)})$. The former is derived by means of the function **Motion_Model_Velocity** which returns the probability of a final position given an initial one and a movement command. The latter is obtained via a scan matching procedure. The approach selected is a point-to-point comparison between the true scan and the scan obtained from the current map for that sample. Both distributions are then multiplied point-wise to obtain the target proposal $\tau$.

The weighing factor $\eta^{(i)}$, the Gaussian mean $\mu_t^{(i)}$ and the covariance $\Sigma_t^{(i)}$ are calculated. The particle weights are also updated. Additionally, the final position estimate for that particle $\ddot{x}_t$ is sampled from the Gaussian. Finally, the map for that particle is updated given the final estimated position $\dot{x}_t$ and the true laser scan.

To check if a resample is necessary, the weights are normalized and $N_eff$ is derived. In the case that this value is smaller than a given percentage of the total amount of particles, a resampling step must be taken. If not, no further steps are required.

# Bibliography

[1] A. Doucet, J. de Freitas, K. Murphy, and S. Rusel. Rao-blackwellized particle filtering for dynamic bayesian networks. In *Conf. Uncertainty Artif. Intell. Stanford, CA*, pages 173–186, 2000.

[2] K. Murphy. Bayesian map learning in dynamic environmnets. In *Conf. Neural Inf. Process. Syst. Denver, CO*, pages 1015–1021, 1999.

[3] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. The MIT Press, 2006.