Robótica Móvel

Robot Localization - Part 3
Introduction to Probabilistic Localization

A - Robot localization using landmarks with EKF

- Exercises based on the work of:
  - Shoudong Huang, Gamini Dissanayake
  - Centre for Autonomous Systems
  - Faculty of Engineering and Information Technology
  - University of Technology, Sydney, Australia
- The main steps are:
  - Create the models (motion/process and sensor/observation models)
  - Create the data (control inputs, landmark locations, simulated observations, ground truth of states for comparison)
  - Prepare data for the EKF (Q, R and P initial covariance matrices, initial state)
  - Run the EKF
    - Uses several auxiliary functions like motionmodel, sensormodel, etc.
  - Plot some results

A1-Motion model for the robot

- Create the motion model function

```
function [xstatet1] = motionmodel(xstate,Vin,Dn,t)
% xstatet1 - the next robot state (pose) (at t+1)
% xstate - the current (x,y,th) pose (at instant t)
% Vin - the linear and angular velocities (V,w)
% Dn - the uncertainty (errors) in velocities (dV, dw)
% t - the sampling time
```

- Suggestion for a solution →

```
%nonlinear discrete time dynamic system motion model
function [xstatet1] = motionmodel(xstate,Vin,Dn,t)
xstatet1(1)=xstate(1)+(Vin(1)+Dn(1))*t*cos(xstate(3));
xstatet1(2)=xstate(2)+(Vin(1)+Dn(1))*t*sin(xstate(3));
xstatet1(3)=xstate(3)+(Vin(2)+Dn(2))*t;
end
```

A2-Observation model (range and bearing)

- Create the sensor/observation model function

```
function [z1]=sensormodel(xym, xstate1, ndnphi)
% z1 - matrix of new sensor data
% xym - vector of landmark position
% xstate1 - vector of robot pose (state)
% ndnphi - vector of uncertainties in measurement (observation noise)
```

- Suggestion for a solution →

```
%nonlinear measurement equation sensor model
function [z1]=sensormodel(xym, xstate1, ndnphi)
z1(1)=sqrt((xym(1)-xstate1(1))^2+(xym(2)-xstate1(2))^2)+ndnphi(1);
z1(2)=atan2(xym(2)-xstate1(2),xym(1)-xstate1(1))-xstate1(3)+ndnphi(2);
end
```

A3-Create data for the simulation

- Analise and edit the provided file **Generate_data_EKF_localization_v3.m**
  - two cases are offered: simple trajectory with 4 steps, or 100 steps circular trajectory (data will be stored in **data/** or **data100/** directories, respectively)
  - Set noise level setting (σ_V, σ_ω,σ_r, σ_ϕ)
  - Set Landmark coordinates
  - Set Control inputs
    - Add noise to control input to simulate error in real system
  - Generate ground truth for robot state (for comparison in the end)
  - Generate observed data
    - The example generates data only for **two** simultaneous landmarks to simplify
    - This can be adapted for other scenarios
- After possibile editions, run the file to generate the data for simulation

A4-The main EKF function

```
function [xstate_t1,P_t1] = ekf(xstate_t,P_t,control_t,obs_t1,landmarkxym,Delta_T,Q,R)
% xstate_t1 -  new calculated state (pose)
% P_t1 - new calculated covariance matrix (uncertainty) of the new state
%
% xstate_t - state at instant t (current)
% P_t - Covariance matrix at instant t (current)
% control_t - current control signal (u)
% obs_t1 - current sensor observation (z)
% landmarkxym - landmarks xy coordinates (to be used in sensor model)
% Delta_T - sampling interval (time)
% Q - input (velocity) noise covariance matrix
% R - sensor (observation) noise covariance matrix
```

- Main code provided in file **ekf.m**
- This file uses the following auxiliary functions
  - motionmodel
  - sensormodel
  - jacobi (to calculate the specific jacobian matrices for this process)
  - GetCov (to get points to plot uncertainty ellipses around poses)

Auxiliary functions for EKF

- Calculate jacobians→

```
function  [H]= jacobi(xym,x,y)
a = (x-xym(1))/(sqrt((x-xym(1))^2+(y-xym(2))^2));
b = (y-xym(2))/(sqrt((x-xym(1))^2+(y-xym(2))^2));
c = 0;
d =
-(y-xym(2))/(((y-xym(2))^2/(x-xym(1))^2+1)*(x-xym(1))^2);
e = 1/(((y-xym(2))^2/(x-xym(1))^2+1)*(x-xym(1)));
f = -1;
H = [a b c
     d e f];
```

- Get points to plot error ellipses→

```
function [cv]=GetCov(P,x,y)
s=1;
k=40;   % original: k=20, changed by zhan
if trace(P)<1e-5, r=zeros(2,2);
else
   %P
   r=real(sqrtm(P));
end
for j=1:k+1
 q=2*pi*(j-1)/k;
 cv(:,j)=s*3*r*[cos(q); sin(q)]+[x;y];
end
```

Illustration of results for a simple case

- 4 beacons with two of them made as always visible
- Observation model uses both range and bearing
- 5 robot poses (and moments of input control)
- Uncertainties were made all equal to 0.1 (large, mainly for angular entities)
- Uncertainty ellipses show the limits of the covariance of the localization
- red - ground Truth
- blue - estimated localization

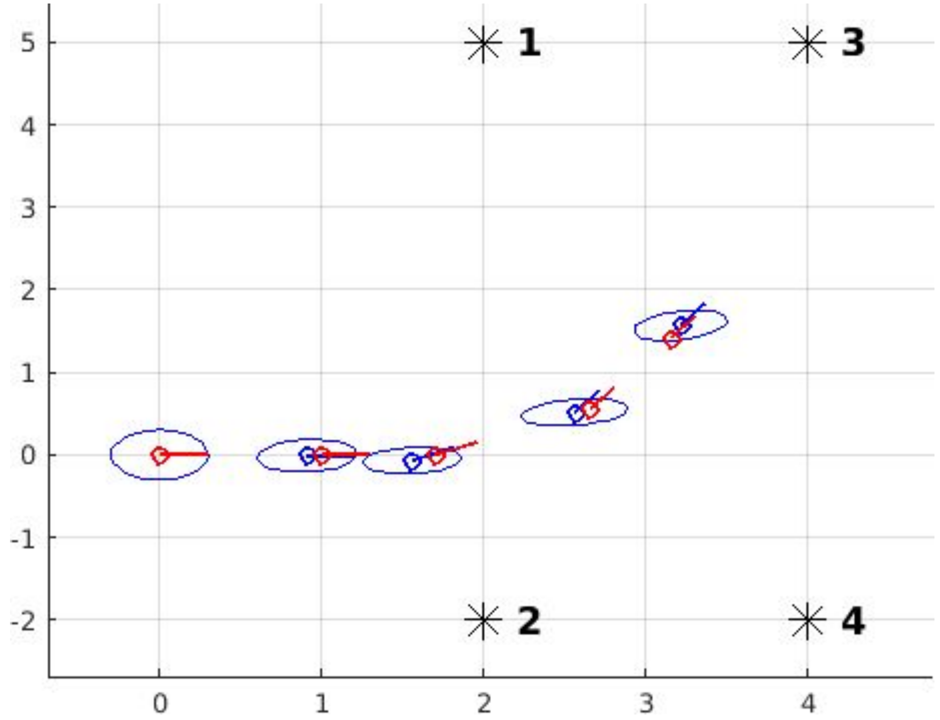
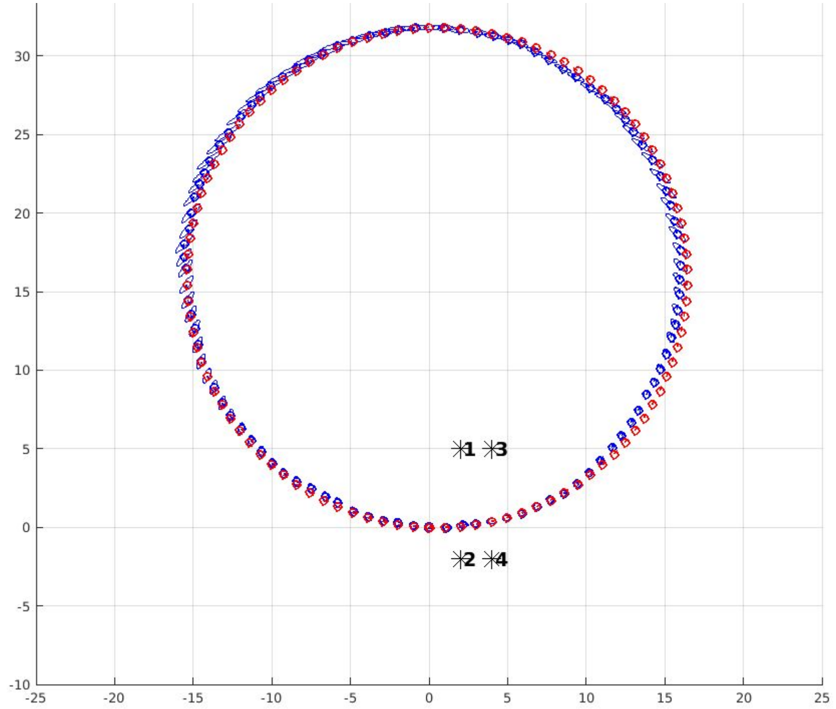


Illustration of results for a longer trajectory

- Same conditions as earlier...
- ...but using 100 poses along a uniform circular motion!
- Uncertainty ellipses now show larger deviations in some points
- In some parts landmarks are very far away and accuracy degrades!
  - How could results be improved?
- red - ground Truth
- blue - estimated localization



A5- Adapt program to new situations

- Try the following options to improve results:
  - Reduce uncertainty in angular variables from 0.1 to 0.05
  - Relocate one beacon (eg. number 1) for better results.

- Compare EKF localization with absolute localization using two beacons with the same angular uncertainty
  - Plot the (x,y) location after 2 beacons used in absolute localization using the robot real orientation

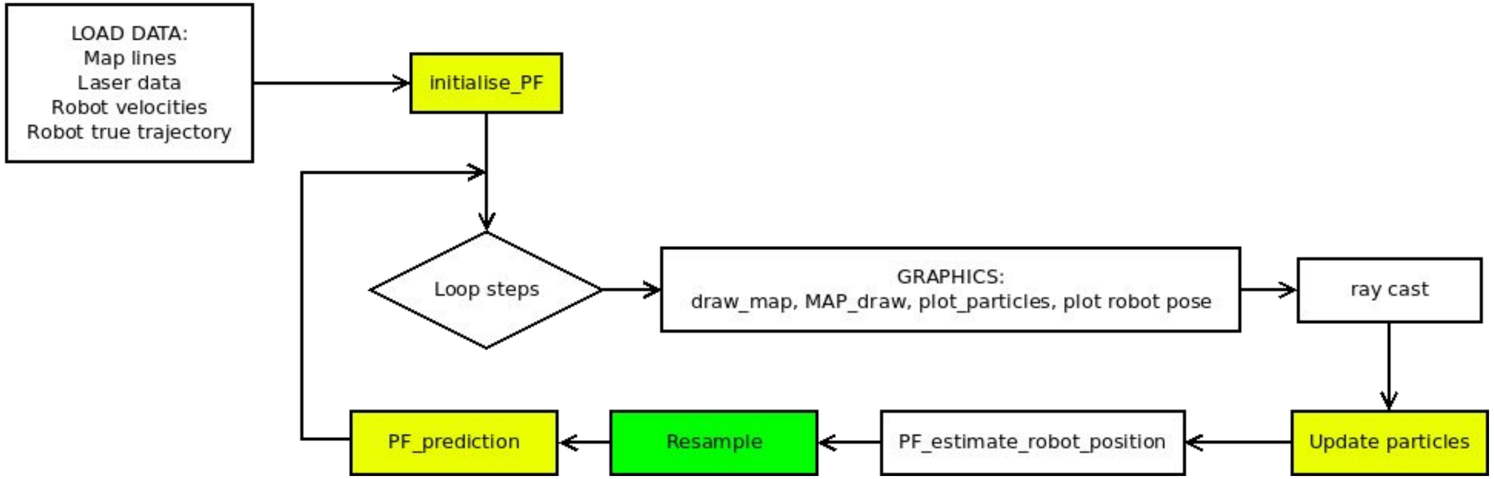
B- Particle Filter Localization

- Exercises based on the work of:
  - Shoudong Huang, Gamini Dissanayake
  - Centre for Autonomous Systems
  - Faculty of Engineering and Information Technology
  - University of Technology, Sydney, Australia

- Objectives and steps of the exercise
  - Understand the flow of the procedure
  - Use provided functions from file and create the remainder required functions
  - Run the exercise and observe the localization progress
  - Modify some parameters of the problem

Main steps of the Particle Filter simulation program

- Flow of program **Robot_Localization_PF_Scan_v3.m**



- Some data can be recreated with the function **Generate_data_PF_localization_v3.m**

B1 - Initialise Particle Filter

Function to initialise the Particle Filter

```
function particle = initialise_PF(n, pose, r, angle, l_num)
% n = Number of particles
% pose = x,y,theta -- centre pose of particles
% r = radius of random particles
% angle = degree of difference from Robot orientation
% l_num = number of laser scans to use from each particle
% → function provided
```

The return variable is a convenient structure

```
% pos = [x,y,theta]
% weight = weight of particle
% llh = likelihood of particle
% laser = distances of each laser beam cast from particle, -90 to 90
```

Auxiliary functions

```
function [r] = random_uniform(a,b)
      r = a + (b-a).*rand(1,1);
end


function valid_particle = particle_validation(particle_x,particle_y)
% this is to check whether the particle is valid or not
% based on the map information -- depends on the map used
valid_particle = 1;
% invalid if the particle is not inside the big rectangle
if particle_x<1.4 || particle_x>5.2 || particle_y<-5.1 || particle_y>5.7
   valid_particle = 0;
% invalid if the particle is inside the small rectangle
elseif particle_x<3.5 && particle_x>2.5 && particle_y<-1.7 && particle_y>-3.4
   valid_particle = 0;
end
```

Other auxiliary functions - 1

```
function draw_map()
% draw the occupancy grid map → function provided


function MAP_draw(LineData, color)
   hold on;
   for j = 1:size(LineData,1)
       plot(LineData(j,1:2),LineData(j,3:4), color)
   end
   hold off;
end


function [shortest_distance, x_found, y_found] =find_distance(max_range,linedata, x3, y3, phi)
% this function compute the intersection of a laser beam with all the line
% segments and find the intersection with shortest distance → function provided

function newparticle = PF_prediction(particle, vel, tr, sig_v, sig_w, time)
% Prediction using the process model → function provided
% vel = velocity
% tr = turn rate
% sig_v = variance of velocity noise
% sig_w = variance of turn rate noise
% time = timestamp
```

Other auxiliary functions - 2

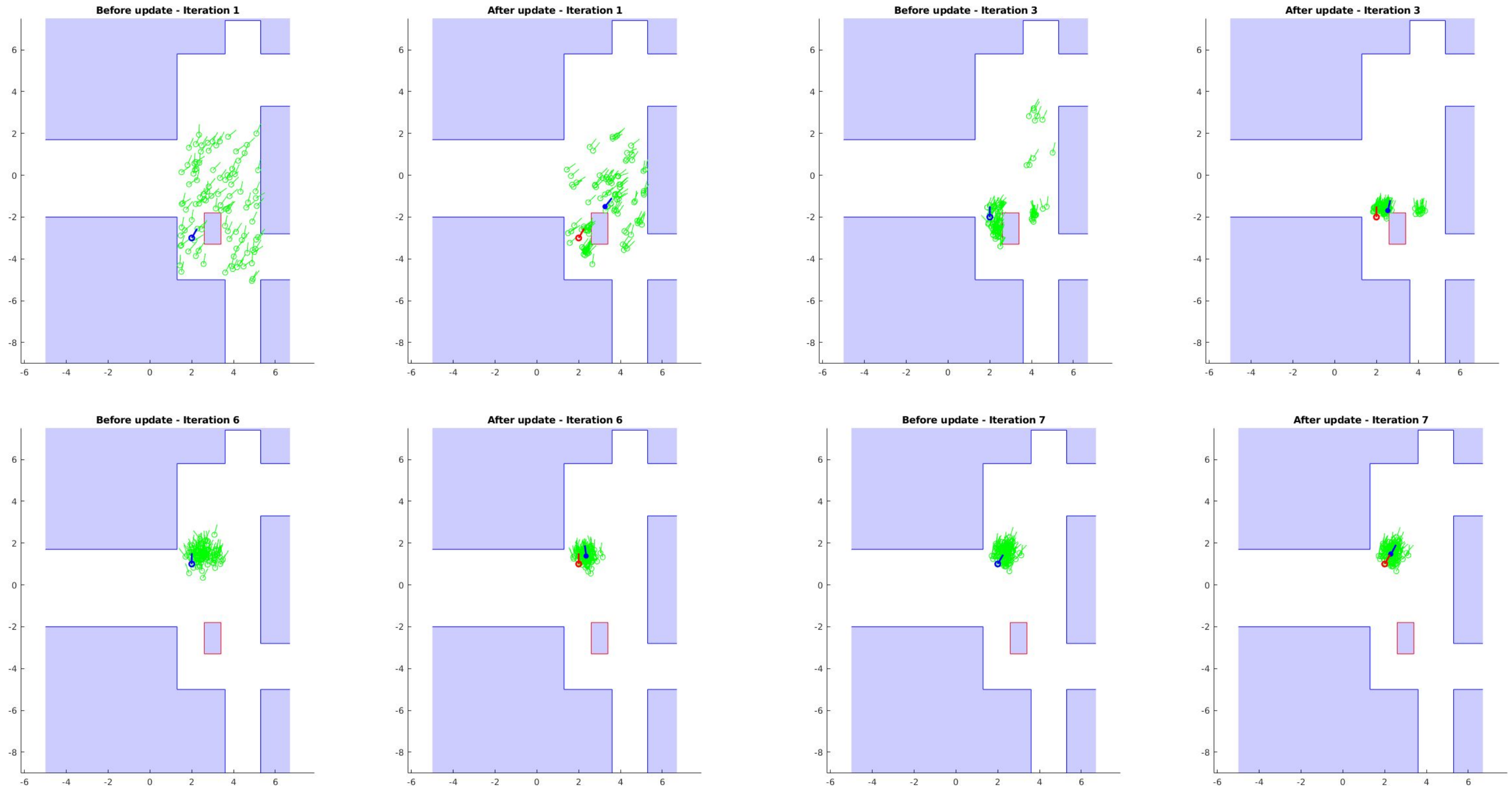
```
function particle = update_particles(particle, robot, sigma)
% → function provided

function robot_estimate = PF_estimate_robot_position(particle, check)
% this function is to estimate the robot pose using the particles → function provided
% check = 1 -- use the weighted sum of all the particles
% otherwise (e.g. check=0) -- use the particle with the largest weight

function [resampledParticles] =resample_particles(particle,method,var_xy,var_angle)
% resample of the particles based on their weights → function provided
% particles with higher weights will get more copies
% particle with very low weights may be eliminated
%
% if method = 1, number of copies of each particle is proportional to its weights
% otherwise, a bit more random
%
% after resampling, also add some noises on the particles
%
% var_xy -- variance of noises to be added to x and y
% var_angle -- variance of noises to be added to angle

 function r = random_normal(a,b)
    r = a + b.*randn(1,1);
 end
```

Example of results obtained - 4 sample iterations



Some adjustable parameters and their current defaults

- Number of particles
  - 100
- Number of lidar beams (to generate the data to use)
  - Laser jump = 30º (angular resolution of laser beams)
- Uncertainties of variables (control inputs and measurement)
  - % variance of control noises
    - sig_v=0.2;
    - sig_w=0.2;
  - % variance for laser noise
    - sig_laser = 0.01;
- Some of these parameters can be adjusted to improve results or speed.