**Simultaneous Localization and Mapping**

**A General Approach to Different Methods**

Submitted by:

*Melikcan Türkdemir*

*Uğur Can Kozan*

*Veysel Erçağlar*

*Yunus Atahan*

Project Supervisor: Assistant Prof. Dr. Deniz Gençağa

Faculty of Engineering

Antalya Bilim University

Spring 2019

**TABLE OF CONTENTS**

|  |  |
| --- | --- |
| ABSTRACT…………………………………………………………………….. | 2 |
| 1 INTRODUCTION   …………………………………………………………… | 3 |
| 2 MOTIVATION …………………………………...………………………….. | 3 |
| 3 METHODOLOGY……….………………………………………………........  3.1 The General Methods…..............................................................................  3.1.1 Kalman Filter...............................................................................  3.1.2 Extended Kalman Filter...............................................................  3.1.3 Particle Filter…................................................................................. 4 CONCLUSION AND CHALLENGES……………………………………….. | 4  4  4  6  13  22 |
| REFERENCES ………………………………………………………………….. | 23 |

**ABSTRACT**

Simultaneous localization and mapping (SLAM) is a method that is used in artificially intelligent robot for an autonomously directed experience. SLAM became significantly popular and a concrete research field in last decades due to its powerfulness in solving many of the problems that are related with autonomous robotics. Since SLAM algorithms first appearance, many SLAM have been created to apply SLAM in real-life scenario such as self-driving car. There are many approaches for solving SLAM and have plusses and drawbacks. The aim of this paper is to compare and implement different SLAM algorithms from scratch.

**1 INTRODUCTION**

Robots in millennium era were always popular. They were popular among both users and researchers. In mobile robots, self-driving or observing from outside and processing inside were important. Under heavy research years, Simultaneous Localization and Mapping (SLAM) became extremely popular among researchers. SLAM is a method that on an unknown location, the agent is creating a map concurrently keeping the data of agent’s location. This technique allows a robot to behave like an intelligent being.

SLAM is widely used in self-driving cars, and robots that built to make investigation on unknown places to people (Such as MARS). SLAM is preferred because with no prior knowledge robots are still making good progress. There are multiple SLAM algorithms on literature that are beneficial in particular case or not effective. Introduced algorithms for SLAM are as Kalman SLAM, EKF SLAM, FAST SLAM, L-SLAM, GraphSLAM, LSD-SLAM, S-PTAM, ORB-SLAM, MonoSLAM, CoSLAM. There are other algorithms used for SLAM but in this paper, we will try to focus on three of them. At the end of this paper, the implementations will show their comparisons in terms of their efficiency, run time complexity and properness.

**2 MOTIVATION**

SLAM is a hot topic in the literature and all of the content demonstrates diversity. As stated in introduction, there multiple methods. The whole process of implementing a SLAM algorithm brings particular requirements within itself. Understanding of each algorithm

One of the main motivation research is efficiency, in terms of run time complexity. In a continuous calculation, and critical-state, a poorly working algorithm cannot be used. In this paper, we are diving into the backgrounds of algorithms by implementing and comparing.

Secondly, the properness. Properness is substantial in terms of usage. In the literature, there is a reason for different approaches. Linear cases, non linear cases, different distributions etc. In the further sections, we will be focusing on which algorithm is beneficial for which case and is giving better results.

**3 METHODOLOGY**

**3.1 The General Methods**

We approached with three different algorithms that are stated in upcoming sections.

*3.1.1 Kalman Filter*

In [statistics](http://www.wikizero.biz/index.php?q=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvU3RhdGlzdGljcw" \o "Statistics) and [control theory](http://www.wikizero.biz/index.php?q=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvQ29udHJvbF90aGVvcnk), Kalman filtering, also known as linear quadratic estimation (LQE), is an [algorithm](http://www.wikizero.biz/index.php?q=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvQWxnb3JpdGht" \o "Algorithm) that uses a series of measurements observed over time, containing [statistical noise](http://www.wikizero.biz/index.php?q=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvU3RhdGlzdGljYWxfbm9pc2U" \o "Statistical noise) and other inaccuracies, and produces estimates of unknown variables that tend to be more accurate than those based on a single measurement alone, by estimating a [joint probability distribution](http://www.wikizero.biz/index.php?q=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvSm9pbnRfcHJvYmFiaWxpdHlfZGlzdHJpYnV0aW9u" \o "Joint probability distribution) over the variables for each timeframe. The filter is named after [Rudolf E. Kálmán](http://www.wikizero.biz/index.php?q=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvUnVkb2xmX0UuX0slQzMlQTFsbSVDMyVBMW4" \o "Rudolf E. Kálmán), one of the primary developers of its theory.

The Kalman filter has numerous applications in technology. A common application is for [guidance, navigation, and control](http://www.wikizero.biz/index.php?q=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvR3VpZGFuY2UsX25hdmlnYXRpb24sX2FuZF9jb250cm9s" \o "Guidance, navigation, and control) of vehicles, particularly aircraft and spacecraft. Furthermore, the Kalman filter is a widely applied concept in [time series](http://www.wikizero.biz/index.php?q=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvVGltZV9zZXJpZXM" \o "Time series)analysis used in fields such as [signal processing](http://www.wikizero.biz/index.php?q=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvU2lnbmFsX3Byb2Nlc3Npbmc" \o "Signal processing) and [econometrics](http://www.wikizero.biz/index.php?q=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvRWNvbm9tZXRyaWNz). Kalman filters also are one of the main topics in the field of robotic motion planning and control, and they are sometimes included in [trajectory optimization](http://www.wikizero.biz/index.php?q=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvVHJhamVjdG9yeV9vcHRpbWl6YXRpb24" \o "Trajectory optimization).

**Code**

Code is written with help of the tutorials [4][5][6] and code below has two essential parts one is generic source function and the other part is the testing part.

import numpy as np

from kalman\_filter import predict

measurements = [0, 1.1, 1.9, 2.5, 3.7, 4.9, 6]

x = np.zeros((2,1)) # initial state (location and velocity)

P = np.eye(2,2)\*1000# initial variance

u = np.zeros((2,1)) # external motion

F = np.array([[1., 1.], [0, 1.]]) # next state function

H = np.array([[1., 0.]]) # measurement function

R = np.array([[1.]]) # measurement variance

for m in measurements:

print("{:6.4f}".format(\*predict(x,u,m,F,P,R,H)[0][0]),"\t", measurements.index(m))

import numpy as np

#x: initial state

#u: external input

#z: measurement

#F: next state matrix

#P: initial variance

#R: Measurement variance

#H: Measurement function matrix

#Q: Process variance

def predict(x, u, z, F, P, R, H=None, Q=None):

#INITIALIZATION

i\_p=np.eye(\*P.shape)

if H is None:

H=np.ones(x.shape)

if Q is None:

Q=np.zeros(P.shape)

#PREDICTION

x\_n=np.add(np.matmul(F, x), u)

P=np.matmul(np.matmul(F, P), F.transpose())+Q

#MEASUREMENTS

z\_n=np.matmul(H, x\_n)

err\_z\_z\_n=np.subtract(z,z\_n)

h\_t=H.transpose()

Knum=np.matmul(P, h\_t)

Kden=np.add(np.matmul(np.matmul(H, P), h\_t),R)

K=np.matmul(Knum,np.linalg.inv(Kden))

#UPDATE

x\_n=np.add(x\_n, np.matmul(K, err\_z\_z\_n))

p\_n=np.matmul(np.subtract(i\_p, np.matmul(K,H)),P)

return x\_n, p\_n

**Output**

0.0000 0

1.0995 1

1.8991 2

2.4988 3

3.6982 4

4.8976 5

5.9970 6

*3.1.2 Extended Kalman Filter*

One of the basic answers for the SLAM was offered by Cheeseman and Smith who processed the EKF to mutually represent the landmark position with the model[1]. It is a class of algorithms that uses Extended Kalman Filter for SLAM problem. EKF is used to estimate the pose of robot and position of landmarks in the map robot moves. Extended Kalman Filter steps is as follows:

* State Prediction:

Estimate new position of the robot

* Measurement Prediction:

Predicting the observation

* Measurement:

Getting real observation with sensors

* Data Association:

Check the difference between predicted observation and real observation that gathered with sensors

* Update:

Change current state (position) of the robot to next state according to the estimation made in data association step.

**EKF-SLAM Implementation**

**Preliminaries:**

xk: The state vector describing the location and orientation of the vehicle.

uk: The control vector, applied at time k-1 to drive the vehicle to a state xk at time k.

mi: A vector describing the location of the *ith*landmark whose true location is assumed time invariant.

zik: An observation taken from the vehicle of the location of the *ith* landmark at time k. When there are multiple landmark observations at any one time or when the specific landmark is not relevant to the discussion, the observation will be written simply as zk.

Also:

X0:k = {x0, x1, … , xk} = {X0:k-1, xk} : The history of vehicle locations.

U0:k = {u1, u2, … , uk} = {U0:k-1, uk} : The history of control inputs.

m = {m1, m2, … , mk} : The set of all landmarks.

Z0:k = {z1, z2, … , zk} = {Z0:k-1, zk} : The set of all landmark observations.

**Vehicle Motion:**

where f(.) models vehicle kinematics and where wk are additive, zero mean uncorrelated Gaussian motion disturbances with covariance Qk.

**Observation Model:**

where h(.) describes the geometry of the observation and where vk are additive, zero mean uncorrelated Gaussian observation errors with covariance Rk.

**The mean:**

**Covariance:**

**Time Update:**

where f is the Jacobian of f evaluated at the estimate

**Observation Update:**

where

and where h is the Jacobian of h evaluated at and

Samsuri et al. points that the runtime complexity of EKF is in worst case ***O(n3)***. [2]

Formally, the algorithm of Extended Kalman Filter as follows:

*Algorithm* **EKF (Problem , Initial Covariance)** *returns the corresponding updated data***{** *Start with Initial Covariance;  
 ObtainedData = Initial Covariance;  
 while true****{*** *Calculate the weights from Initial Covariance;  
 Consider the noise;  
 Get new measurements;  
 ⅋ = Update the state estimations;  
 Calculate the new covariance with obtained ⅋;  
 Guess the new state estimation and covariance for the* ***tt+1*** *step;  
 ObtainedData = Estimation and Covariance;  
 return ObtainedData;*  
 **}  
}**

**Code**

In this section, EKF SLAM method is implemented. It is an imperfect version that should be improved. The python code as follows:

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

# Time

k = [i for i in range(10)]

# State estimates

x = []

# Predicted state

xPred = []

# Prediction error

p = []

# Kalman gain

g = []

# Observations

z = []

x0 = 1000

r = 200

a = 0.90

p0 = 1

x.append(x0)

p.append(p0)

g.append(0)

for i in range(9):

x.append(x[i] \* a)

for i in range(len(x)):

z.append(x[i] + np.random.uniform(-r, r, None))

xPred.append(z[0])

for i in range(1, len(z)):

# Predict

xPred.append(a \* xPred[i - 1])

p.append(a \* p[i-1] \* a)

# Update

g.append(p[i - 1] / (p[i - 1] + r))

xPred[i] = xPred[i] + g[i] \* (z[i] - xPred[i])

p[i] = (1 - g[i]) \* p[i]

print(p)

print(xPred)

print(g)

dataFrame = pd.DataFrame({'x': k, 'State Estimates':x, 'Observations':z, 'Predicted State':xPred})

palette = plt.get\_cmap('Set1')

plt.style.use('seaborn-darkgrid')

num = 0

for column in dataFrame.drop('x', axis=1):

num += 1

plt.plot(dataFrame['x'], dataFrame[column], marker='', color=palette(num), lineWidth=2, alpha=0.8, label=column)

plt.legend(loc=1, ncol=4)

plt.xlabel("Time")

plt.show()

And by running the result we conclude:

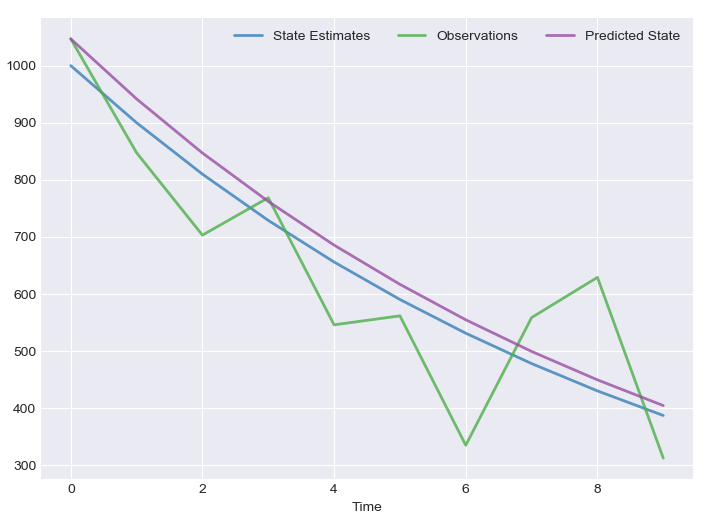


Figure 1 - Measurement

**EKF SLAM – 2nd Approach**

import numpy as np

#x: initial state

#u: external input

#z: measurement

#F: next state matrix

#P: initial variance

#R: Measurement variance

#g: Confidence level for validation gate

#H: Measurement function matrix

#Q: Process variance

def predict(x, u, z, F, P, R, g, H=None, Q=None, round=False):

#INITIALIZATION

i\_p=np.eye(\*P.shape)

if H is None:

H=np.ones(x.shape)

if Q is None:

Q=np.zeros(P.shape)

#PREDICTION

x\_n=np.add(np.matmul(F, x), u)

P=np.matmul(np.matmul(F, P), F.transpose())+Q

#MEASUREMENTS

z\_n=np.matmul(H, x\_n)

err\_z\_z\_n=np.subtract(z,z\_n)

h\_t=H.transpose()

Knum=np.matmul(P, h\_t)

Kden=np.add(np.matmul(np.matmul(H, P), h\_t),R) #S matrix

S\_inv=np.linalg.inv(Kden)

K=np.matmul(Knum,S\_inv) # Filter Gain W matrix

#UPDATE

x\_n=np.add(x\_n, np.matmul(K, err\_z\_z\_n))

p\_n=np.matmul(np.subtract(i\_p, np.matmul(H, K)),P)

#P ESTIMATION CHANGES IN EKF FOR NUMERICAL ROUNDING PROBLEMS

i\_wh=np.subtract(i\_p, np.matmul(K, H))

i\_wh\_t=i\_wh.transpose()

wrwt=np.matmul(np.matmul(K,R),K.transpose())

P\_N=np.matmul(np.matmul(np.matmul(i\_wh, P),i\_wh\_t),wrwt)

e\_sq = np.matmul(np.matmul(err\_z\_z\_n, S\_inv), err\_z\_z\_n.transpose()) # EXTENDED KALMAN FILTER FEATURE

if not e\_sq <= g \*\* 2: # VALIDATION GATE

raise Exception('VALIDATION GATE: MEASUREMENT EXCEEDS EXPECTED LEVELS')

if round:

return x\_n, P\_N

return x\_n, p\_n

import numpy as np

from ekf import predict

import matplotlib.pyplot as plt

MEAN=0

VARIANCE=10

LENGTH=50

t=np.linspace(0,3,LENGTH)

#func=10\*np.sin(2\*np.pi\*t)

func=np.zeros((1, LENGTH))

func=func.reshape((LENGTH,))

measurements=func+np.random.normal(MEAN, VARIANCE, LENGTH)

x = np.zeros((2,1)) # initial state (location and velocity)

P = np.array([[1., 0.],

[0., 0.]])\*10# initial variance

u=np.zeros((2,LENGTH))

u[0] = func # external motion

u[1] = np.full((1,LENGTH),1/LENGTH)

F = np.array([[1., 0.],

[0., 0.]]) # next state function

H = np.array([[1., 0.]]) # measurement function

R = np.array([[1.]])\*VARIANCE # measurement variance

x\_n=x

p\_n=P

x\_predict=[]

for i, m in enumerate(measurements):

x\_n, p\_n = predict(x,u[:,[i]],m,F,p\_n,R,10000,H)

x\_predict.append(x\_n[0,0])

plt.figure(figsize=(18,5))

x\_plot=np.arange(LENGTH)

plt.plot(x\_plot, measurements, linestyle='None', marker="x", markersize=5)

plt.plot(x\_plot, x\_predict)

plt.plot(x\_plot, func)

plt.tight\_layout()

plt.show()

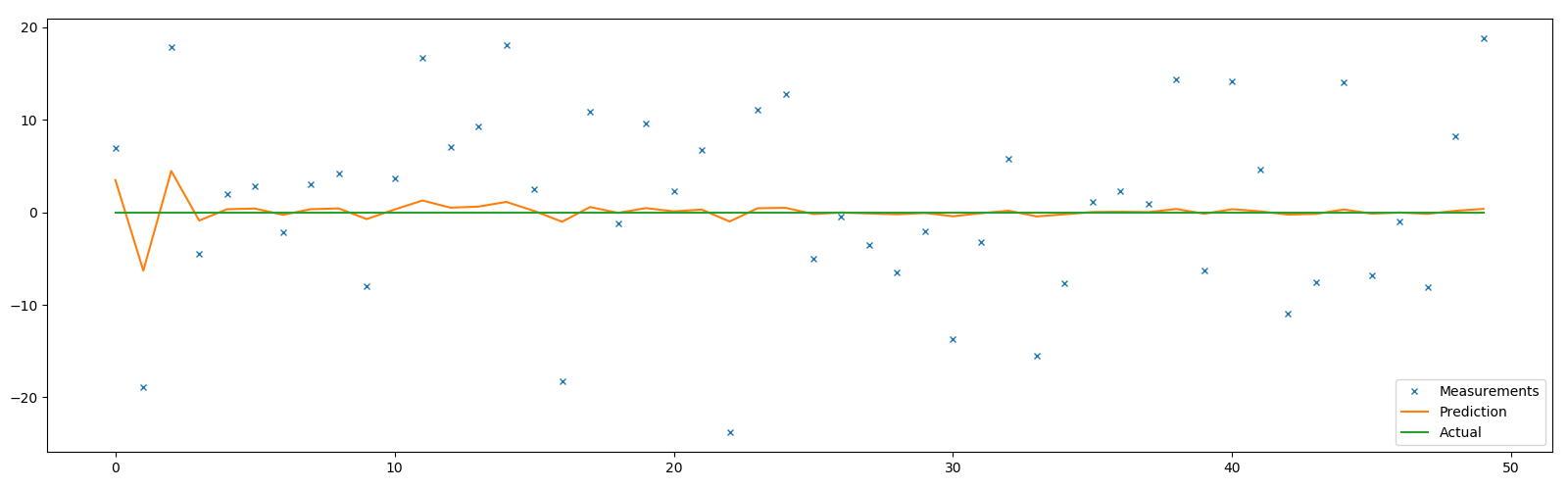
****

Figure 2 f=0, N(0,10)

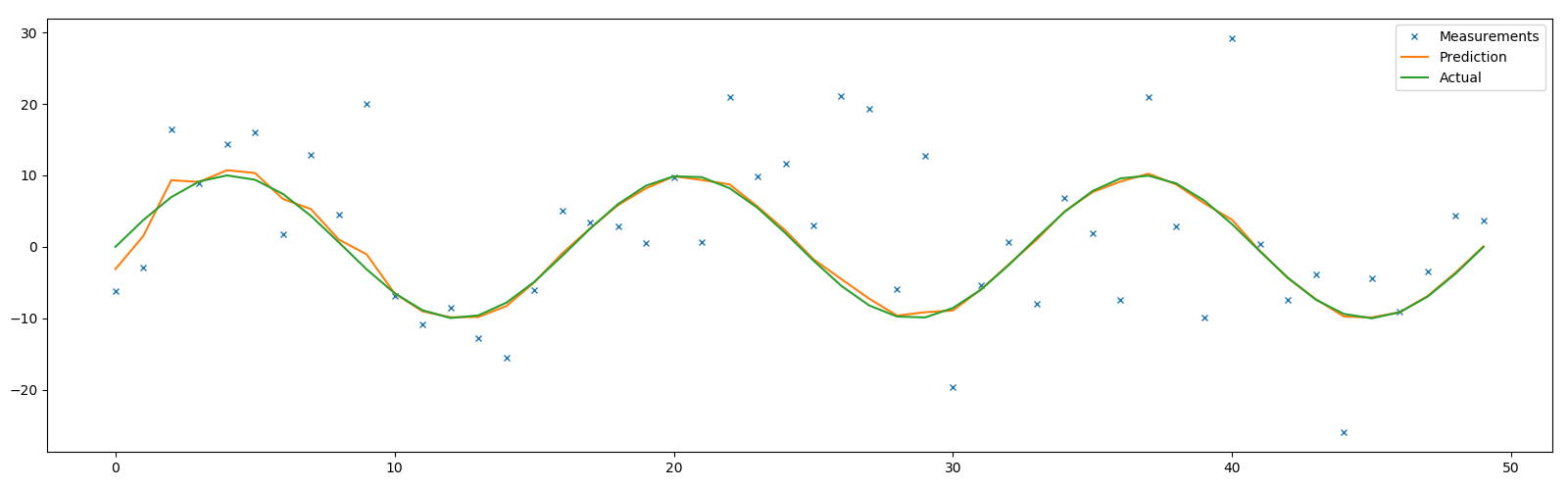
****

Figure 3 f=10sin(2πt), N(0,10)

*3.1.3 Particle Filter*

Particle Filter is a method that computing the posterior behavior in limited Markov Chains within discrete time. In a given time *t,* a state of Markov Chain is xt . Clearly, the state of xt is bounded to state xt-1 under regards of probabilistic law

Another state kt, that will be a stochastic projection of xt . Eventually, it will be generated by probabilistic approach: . In a generalist way of representation of estimation is and the measurement (update) is . Specific Kalman Filters are working under *O(d3*) run time complexity. “d” here is the given dimension space. Kalmans are for the cases where the Gaussian-Linear assumptions are appropriate for estimation. However, particle filters are in a generalist cases of partially unconstrained Markov Chains. The base structure is to estimate the posterior of a set of sample states } or particles. [7] denotes state of sample *i* and range varies between [1,*n*), *n* is the volume of particle filter. Particle Filters are working with the “Survival of the Fittest” concept. Each posterior is denoted with set of “weighted samples” Each particle is distributed randomly initially and their lifespan is decided by their weights. The generic pseudocode as follows:

-**Algorithm Particle Filter(*problem)*** *returns the resulting set of particles{  
 Initiate n many particles at time t=*t0 (Initial Time)  
 *Particle0 =Distribute initiated particles with respect to p(x0)* (Under Gaussian)  
 *While(t >0) {* Xt= *Create a particle for each previous state’s particle() from prediction  
 Distribute n particles from* Xt  *,each is distributed with probabilistic update )*

*}  
Return the outcome set of particles Xt  
}*

Eventually, a specialized particle filter algorithm will be used for SLAM. That is called Fast SLAM.

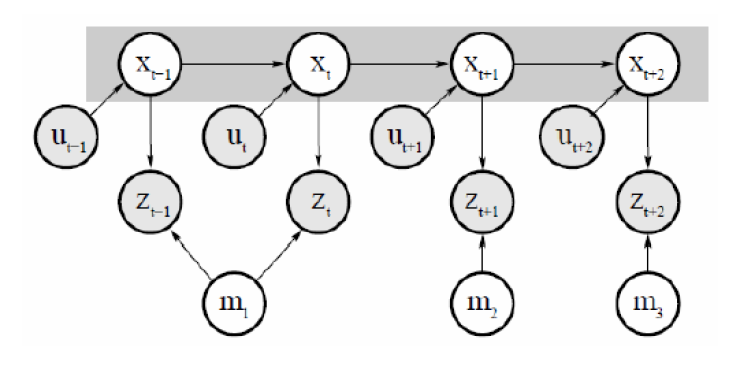


Figure 4 Bayes network for the Fast SLAM problem

**Fast SLAM**

Fast SLAM algorithm is introduced by Montemerlo et al. in 2002 as first successful implementation of Rao-Blackwellised particle filter that could handle large maps or real-world problems. Each landmark is represented by 2x2 EKF, therefore each particle must maintain M individual EKFs. In total, there are N·M EKFs, where M is the total number of particles in the particle filter and N is the total number of landmarks.

**Key Steps of Fast SLAM 1.0**

Fast SLAM algorithm draws samples according to standard odometry model being used to localization. It extends the path posterior by sampling a new pose for each sample.

In the next step, it computes the importance weight:

Q: measurement covariance

z: current observation

ẑ: expected observation (calculated for each individual)

As last step, it updates the belief of observed landmarks using the EKF update rule, then resamples using the standard resampling operation.

**Computational Complexity Fast SLAM Implementation**

Update robot particles: *O(N)*

Incorporate an observation into Kalman filters: *O(N log M)*

Resample particle set: *O(N log M)*

Total: *O(N log M)*

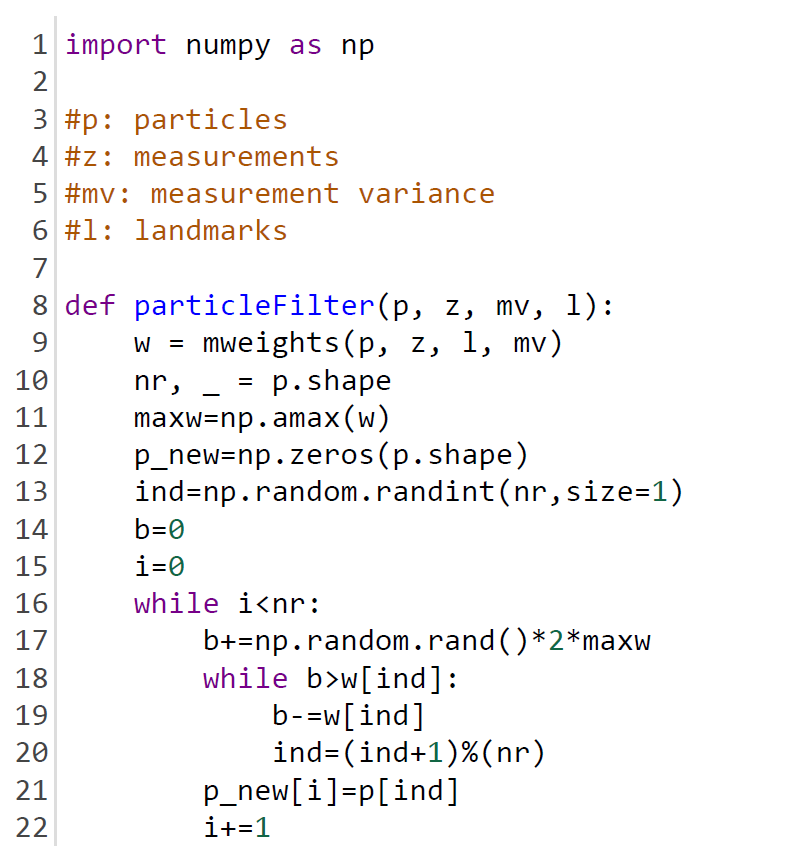
(*where N is the number of particles and M is the number of map features*)

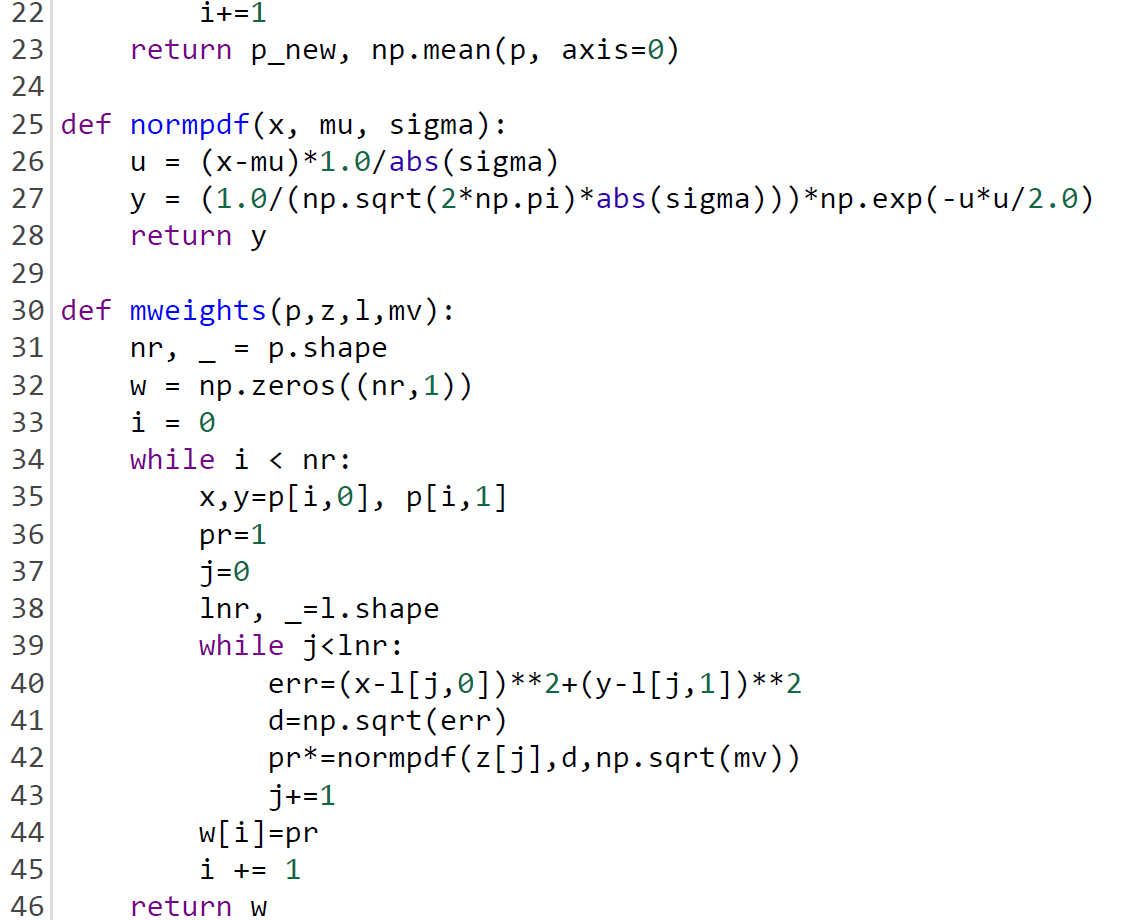
**Fast SLAM 2.0**

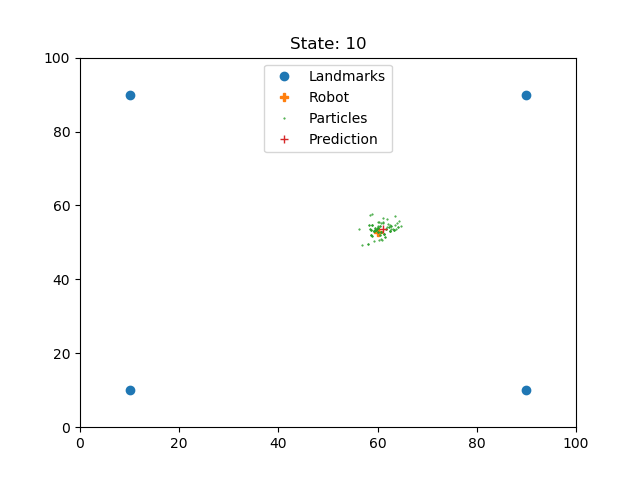
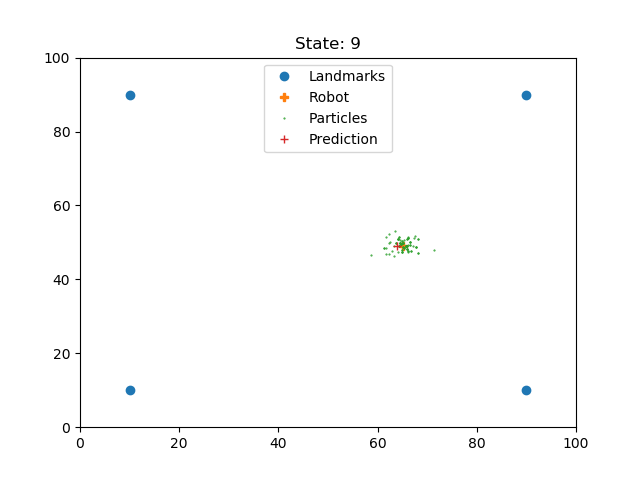
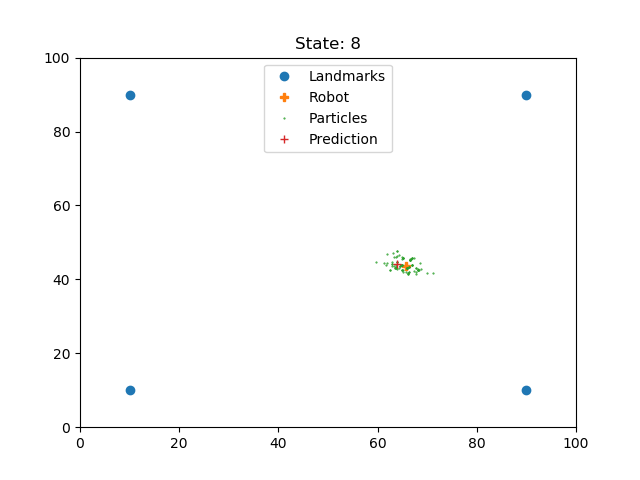
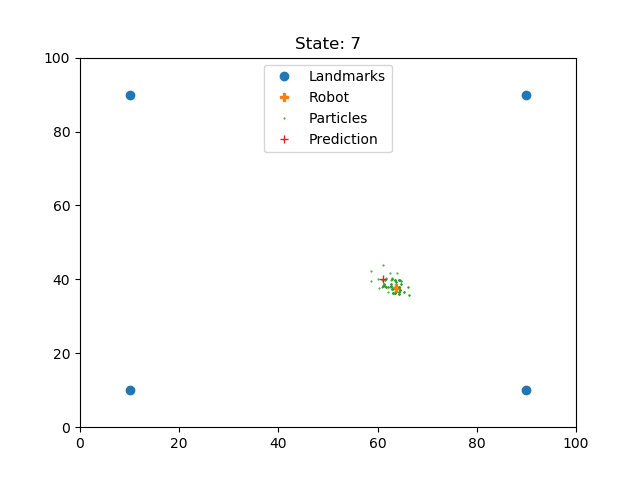
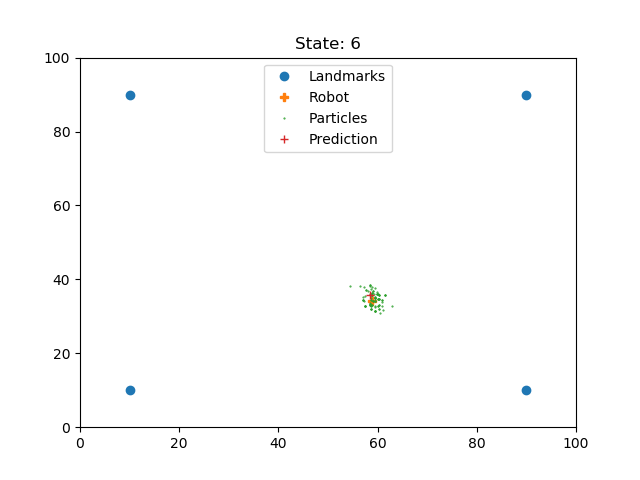
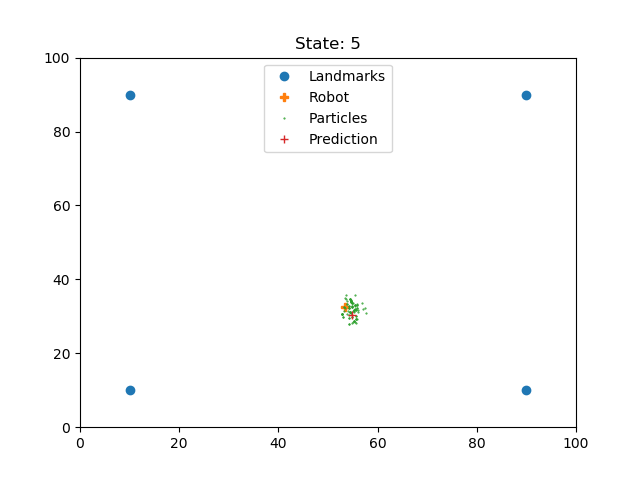
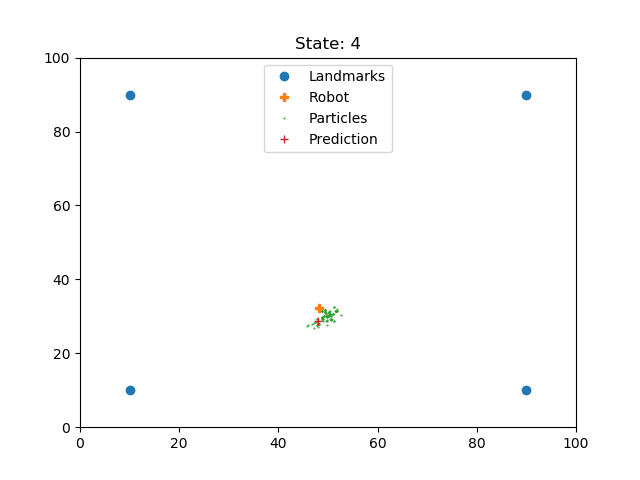
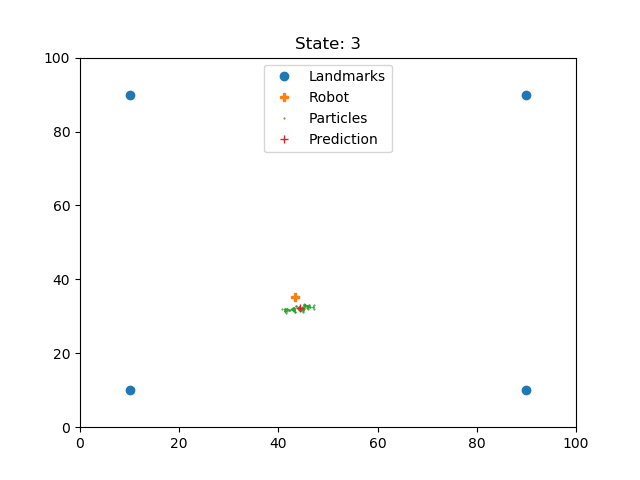
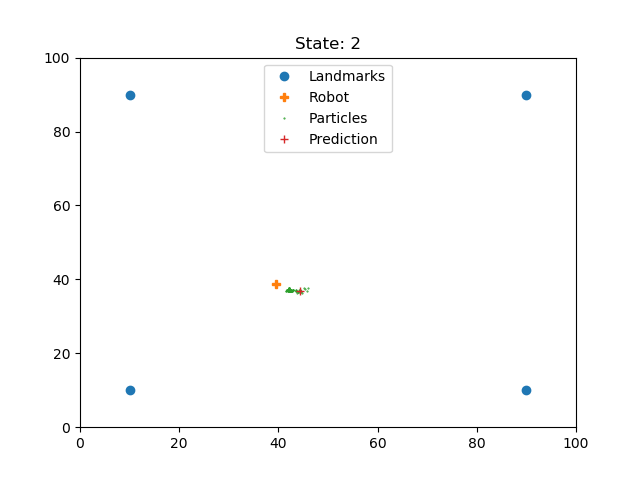
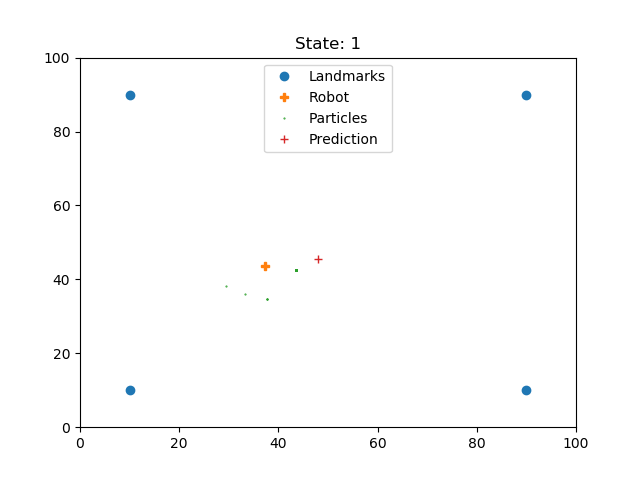
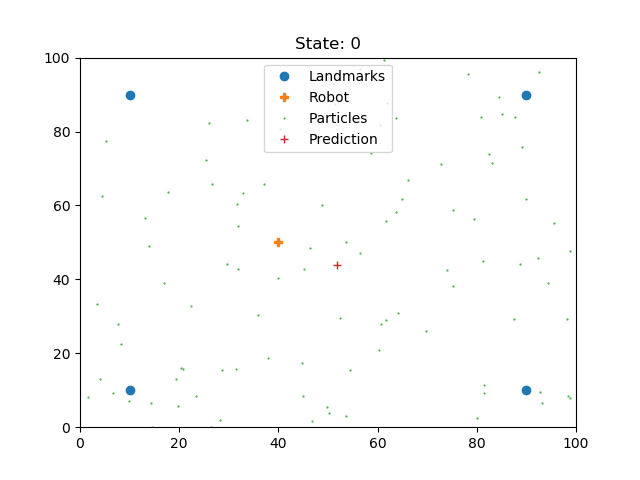
Second iteration of Fast SLAM proposed by Montemerlo et al. in 2003, which considers the measurements during the sampling.

This leads to proposal distribution being more peaked around the true state where the system is in. As a result, less samples are needed. Compared to Fast SLAM 1.0, Fast SLAM 2.0 is more robust and accurate, however it is more complex.

**Implementation of Particle Filter SLAM**





Results of above implementation with respect to states:****

**4 CONCLUSIONS AND CHALLENGES**

**REFERENCES**

[1] R. Smith and P. Cheeseman. *On the representation and estimation of spatial uncertainty.* *Intl. J. of Robotics Research*, 5(4):56–68, 1987.

[2] Samsuri et al. *Computational Cost Analysis of Extended Kalman Filter in Simultaneous Localization and Mapping Problem for Autonomous Vehicle* 2015

Russell, S. and Norvig, P. (2016). *Artificial Intelligence: A Modern Approach*. 3rd ed. Boston: Pearson.

[3] Jaiswal, P., 2018, *Sensor Fusion — Part 2: Kalman Filter*, Retrieved from <https://towardsdatascience.com/sensor-fusion-part-2-kalman-filter>

[4] Kleeman, L., 2017, *Understanding and ApplyingKalman Filtering*, Retrieved from http://biorobotics.ri.cmu.edu/papers/sbp\_papers/integrated3/kleeman\_kalman\_basics.pdf

[5] LAARAIEDH, M., 2018, *Implementation of Kalman Filter with Python Language*, Retrieved from <https://arxiv.org/ftp/arxiv/papers/1204/1204.0375.pdf>  
  
[6] Durrant-Whyte, Hugh and Bailey, Tim, (2006). “*Simultaneous Localisation and Mapping (SLAM): Part I The Essential Algorithms”*, Robotics & Automation Magazine. 13.

[7] Thrun S. Particle Filters in Robotics, *In Proceedings of Uncertainty in AI (UAI)*

Carnegie Mellon University-Computer Science Department, Section 2, 2002.

[8] Çulha1 U., Turan B. *Particle Filter Based Fast Simultaneous Localization and Mapping.* Computer Engineering Department, Bilkent University, -