Simultaneous Localization and Mapping: A General Approach to Different Methods  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Progress Report II \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Veysel Erçağlar Yunus Atahan Uğur Can Kozan Melikcan Türkdemir

**I. 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 etc.

**II.METHOD**

**2.1 EKF-SLAM**

In this section, previously written 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

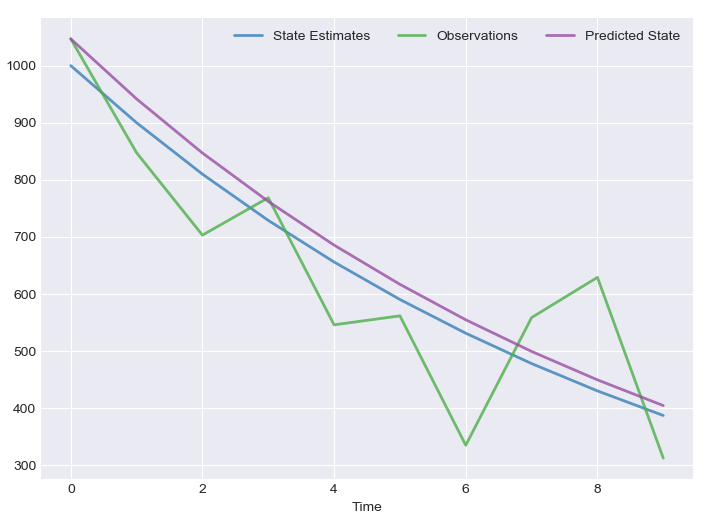
plt.plot(dataFrame['x'], dataFrame[column], marker='', color=palette(num), lineWidth=2, alpha=0.8, label=column)

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

plt.xlabel("Time")

plt.show()

And by running the result we conclude:



**2.2 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()

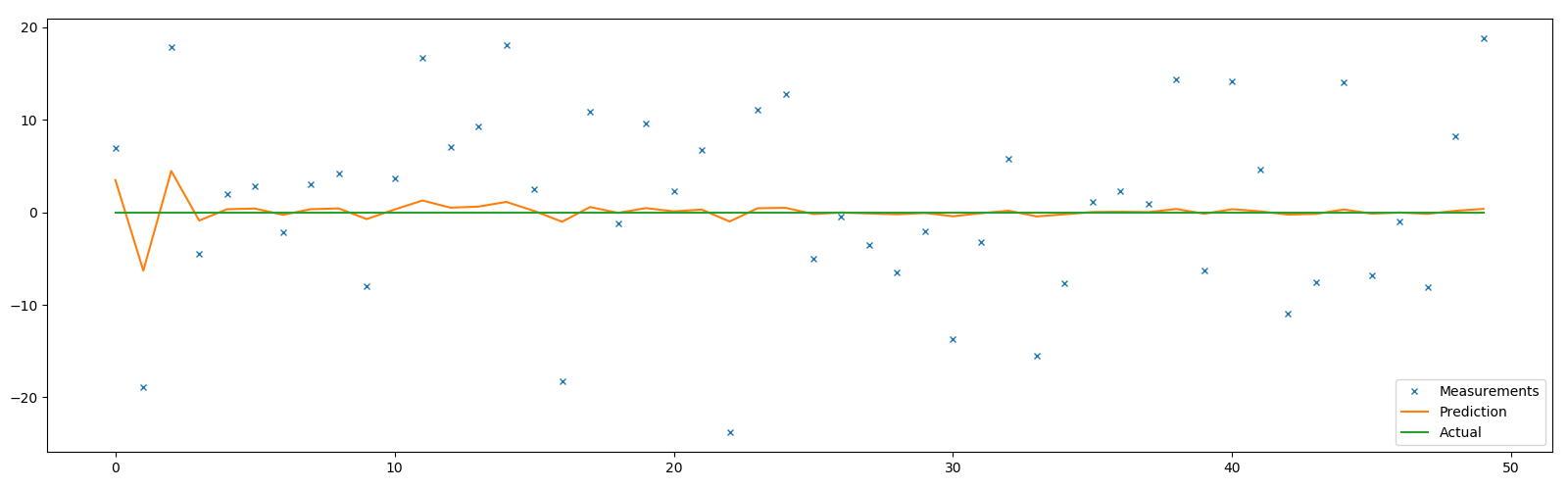
****

Figure 1 f=0, N(0,10)

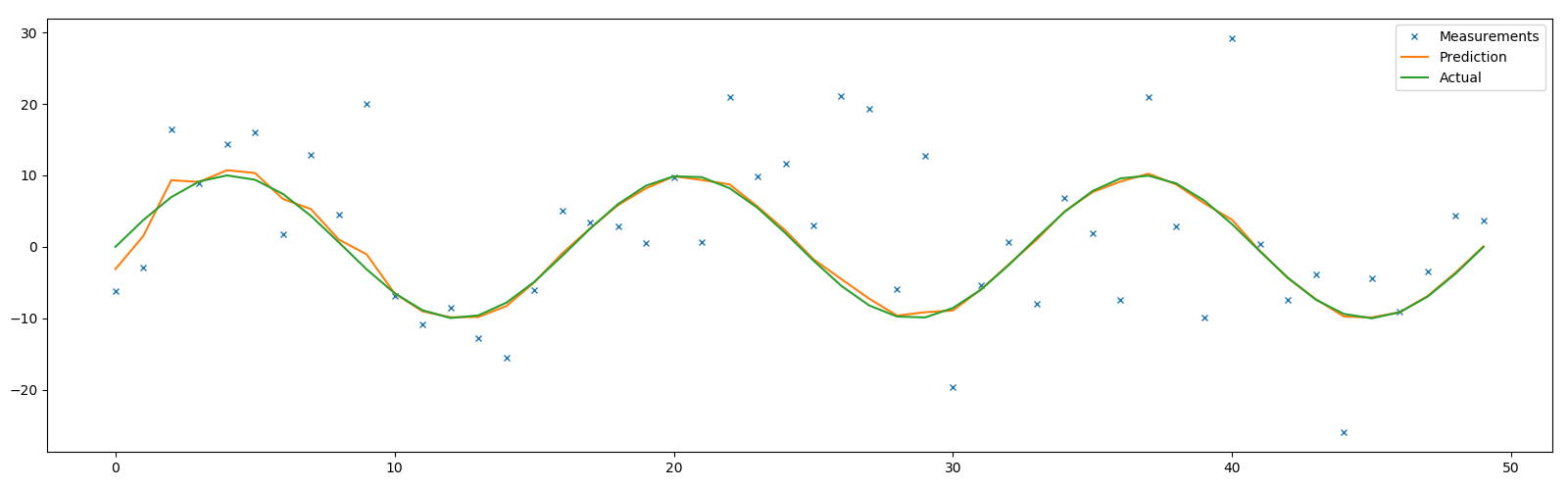
****

Figure 1 f=10sin(2πt), N(0,10)

**2.3 Particle Filter SLAM**

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. [1] 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 particles 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.

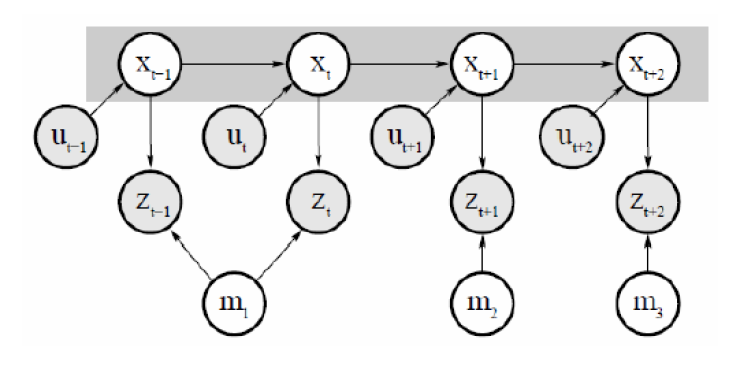


Figure 3 Bayes network for the Fast SLAM problem

**REFERENCES**

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

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

[2] Çulha1 U., Turan B. *Particle Filter Based Fast Simultaneous Localization and Mapping*

Computer Engineering Department, Bilkent University, -