2D-Feature Based EKF SLAM

Simulation Report

EKF SLAM

Done By Rumesh

Done by

Rumesh

Updated on

October 28, 2021

Abstract

Two dimensional **S**imultaneous **L**ocalization **A**nd **M**apping (SLAM) with **E**xtended **K**alman **F**ilter (EKF) is described in this report where hundred landmarks are used for the simulation. EKF SLAM algorithm analyzed with loop closure and without loop closure. The algorithm was implemented using `Python`. The python code is adapted from Atsushi Sakai[2] and modified. Code is attached in Appendix section.

Contents

List of Tables **1**

List of Figures **1**

1 Introduction **2**

2 Model **2**

3 Simulation and Results **6**
- 3.1 Simulation . . . 6
- 3.2 Results . . . 6

Bibliography **8**

Appendix **8**
- .1 Python Code . . . 8

List of Tables

1 The Extended Kalman Filter Algorithm [3] . . . 3
2 The EKF algorithm for the SLAM problem (with known correspondences) [3] . . . 4
3 Simulation Parameters . . . 6
4 Implemeted Python Functions . . . 8

List of Figures

1 EKF-SLAM without Loop Closure I . . . 6
2 EKF-SLAM without Loop Closure II . . . 7
3 EKF-SLAM with Loop Closure . . . 7

Note:
The python code ('EKF_SLAM.py') attached in Appendix Section and the simulation video ('EKF_SLAM.mp4') is attached along with the report repo.

1 Introduction

Kalman filter (KF) is an algorithm that uses a series of data observed over time, which contains noise and other inaccuracies, to estimate unknown variable accurately. It was proposed by Rudolf E. Kálmán in 1960, and became a standard approach for optimal estimation. KF is a real-time, recursive and efficient estimation algorithm in standard (KF), **extended (EKF)**, an unscented (UKF) forms. Standard Kalman filter has optimal kalman gain. To use standard Kalman filter, system must be **linear with gaussian noise** added. In reality, Most of the system in are non-linear. So the the standard Kalman filter will not provide good results for modeling most systems. By linearizing the non linear model able to use the standard Kalman filter to a nonlinear system. So modified EKF comes into role, EKF has near optimal Kalman gain. EKF is sub-optimal estimation technique. There are a couple of different approaches to SLAM but the three main methods are Extended Kalman Filter (EKF), Particle Methods and Graph Based Optimization Techniques (Graph-SLAM). Maps in EKF SLAM are feature based. They are composed of point landmarks. For computational reasons, the number of point landmarks is usually small (e.g., smaller than 1,000). Further, the EKF approach tends to work well the less ambiguous the landmarks. For this reason, EKF SLAM requires significant engineering of feature detectors, sometimes using artificial beacons as features.[3] There are uncertainties in mobile robot motion and measurements. If measurement uncertainties increases over time (assuming no loop closures) SLAM does not provide meaningful result. Loop closing means recognizing an already mapped area. Uncertainties reduces after a loop closure (whether the closure was correct or not). Wrong loop closures lead to filter divergence.[1]

`Python` is used to implement and analyze the EKF-SLAM. *Numpy* `(version 1.20.3)`, *matplotlib* `(version 3.4.2)` *and math library functions* are used and other key functions for EKF are implemented.

2 Model

This report describes the simulation of the 2D Feature based EKF SLAM with mobile robot which achieves loop closure. EKF SLAM modeled as modeled state is both the pose (x, y, θ) and an array of landmarks $[(x_1, y_1), (x_2, x_y), ..., (x_n, y_n)]$ for n landmarks. The covariance between each of the positions and landmarks are also tracked. In our model 100 landmarks are initiated, so 2D state vector (Eq 1 or 3) size expandable is 203. (size calculated by 3+2N where N is Number of landmarks.) The motion model noise and observation noise are gaussian noise, the mean is 0, the covariance is [0,1].

Nomenclatures:

X - 2D state vector, P - State Covariance, R - Process Noise, Q - Sensor Noise, K - Sub optimal Kalman gain, U - Control Input, H - Observation matrice, z - measurement matrice, Δt - Time step, N - Total number of features in MAP

$$X = \begin{bmatrix} x \\ y \\ \theta \\ x_1 \\ y_1 \\ x_2 \\ y_2 \\ \dots \\ x_n \\ y_n \end{bmatrix} \tag{1}$$

$$P = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{x\theta} & \sigma_{xx_1} & \sigma_{xy_1} & \sigma_{xx_2} & \sigma_{xy_2} & \cdots & \sigma_{xx_n} & \sigma_{xy_n} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{y\theta} & \sigma_{yx_1} & \sigma_{yy_1} & \sigma_{yx_2} & \sigma_{yy_2} & \cdots & \sigma_{yx_n} & \sigma_{yy_n} \\ & & & & \vdots & & & & & \\ \sigma_{x_nx} & \sigma_{x_ny} & \sigma_{x_n\theta} & \sigma_{x_nx_1} & \sigma_{x_ny_1} & \sigma_{x_nx_2} & \sigma_{x_ny_2} & \cdots & \sigma_{x_nx_n} & \sigma_{x_ny_n} \end{bmatrix} \tag{2}$$

A single estimate of the pose is tracked over time, while the confidence in the pose is tracked by the covariance matrix P. P is a symmetric square matrix with each element in the matrix corresponding to the covariance between two parts of the system. σ_{xy} represents the covariance between the belief of x and y and which is equal to σ_{yx}. The state can be represented more concisely as follows.

$$X = \begin{bmatrix} x \\ m \end{bmatrix} \tag{3}$$

$$P = \begin{bmatrix} \Sigma_{xx} & \Sigma_{xm} \\ \Sigma_{mx} & \Sigma_{mm} \end{bmatrix} \tag{4}$$

Here the state simplifies to a combination of pose (x) and map (m). The covariance matrix becomes easier to understand and simply reads as the uncertainty of the robots pose (Σ_{xx}), the uncertainty of the map (Σ_{mm}), and the uncertainty of the robots pose with respect to the map and vice versa (Σ_{xm}, Σ_{mx}).
Table 1 summarizes the EKF Algorithm. At each loop, prediction and correction is done.
Table 2 summarizes EKF algorithm for the SLAM problem. Which shows Prediction step in 2-5, Correction steps in 6-14.

1: **Algorithm Extended_Kalman_filter**($\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$):
2: $\bar{\mu}_t = g(u_t, \mu_{t-1})$
3: $\bar{\Sigma}_t = G_t \, \Sigma_{t-1} \, G_t^T + R_t$
4: $K_t = \bar{\Sigma}_t \, H_t^T (H_t \, \bar{\Sigma}_t \, H_t^T + Q_t)^{-1}$
5: $\mu_t = \bar{\mu}_t + K_t(z_t - h(\bar{\mu}_t))$
6: $\Sigma_t = (I - K_t \, H_t) \, \bar{\Sigma}_t$
7: *return* μ_t, Σ_t

Table 1: The Extended Kalman Filter Algorithm [3]

1. Motion Model
The following equations describe the predicted motion model of the robot in case we provide only the control (v, w), which are the linear and angular velocity respectively.

$$F = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5}$$

$$B = \begin{bmatrix} \Delta t cos(\theta) & 0 \\ \Delta t sin(\theta) & 0 \\ 0 & \Delta t \end{bmatrix} \tag{6}$$

$$U = \begin{bmatrix} v_t \\ w_t \end{bmatrix} \tag{7}$$

$$X_{\mathrm{t+1}} = FX_{\mathrm{t}} + BU \tag{8}$$

$$\begin{bmatrix} x_{t+1} \\ y_{t+1} \\ \theta_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} + \begin{bmatrix} \Delta t cos(\theta) & 0 \\ \Delta t sin(\theta) & 0 \\ 0 & \Delta t \end{bmatrix} \begin{bmatrix} v_t + \sigma_v \\ w_t + \sigma_w \end{bmatrix} \tag{9}$$

Notice that while U is only defined by v_t and w_t, in the actual calculations, $+\sigma_v$ and $+\sigma_w$ appear. These values represent the error between the given control inputs and the actual control inputs.

Algorithm EKF_SLAM_known_correspondences($\mu_{t-1}, \Sigma_{t-1}, u_t, z_t, c_t$):

$$F_x = \begin{pmatrix} 1 & 0 & 0 & 0\cdots0 \\ 0 & 1 & 0 & 0\cdots0 \\ 0 & 0 & 1 & \underbrace{0\cdots0}_{3N} \end{pmatrix}$$

$$\bar{\mu}_t = \mu_{t-1} + F_x^T \begin{pmatrix} -\frac{v_t}{\omega_t}\sin\mu_{t-1,\theta} + \frac{v_t}{\omega_t}\sin(\mu_{t-1,\theta}+\omega_t\Delta t) \\ \frac{v_t}{\omega_t}\cos\mu_{t-1,\theta} - \frac{v_t}{\omega_t}\cos(\mu_{t-1,\theta}+\omega_t\Delta t) \\ \omega_t\Delta t \end{pmatrix}$$

$$G_t = I + F_x^T \begin{pmatrix} 0 & 0 & -\frac{v_t}{\omega_t}\cos\mu_{t-1,\theta} + \frac{v_t}{\omega_t}\cos(\mu_{t-1,\theta}+\omega_t\Delta t) \\ 0 & 0 & -\frac{v_t}{\omega_t}\sin\mu_{t-1,\theta} + \frac{v_t}{\omega_t}\sin(\mu_{t-1,\theta}+\omega_t\Delta t) \\ 0 & 0 & 0 \end{pmatrix} F_x$$

$$\bar{\Sigma}_t = G_t\,\Sigma_{t-1}\,G_t^T + F_x^T\,R_t\,F_x$$

$$Q_t = \begin{pmatrix} \sigma_r^2 & 0 & 0 \\ 0 & \sigma_\phi^2 & 0 \\ 0 & 0 & \sigma_s^2 \end{pmatrix}$$

for all observed features $z_t^i = (r_t^i\ \phi_t^i\ s_t^i)^T$ *do*

$j = c_t^i$

if landmark j *never seen before*

$$\begin{pmatrix} \bar{\mu}_{j,x} \\ \bar{\mu}_{j,y} \\ \bar{\mu}_{j,s} \end{pmatrix} = \begin{pmatrix} \bar{\mu}_{t,x} \\ \bar{\mu}_{t,y} \\ s_t^i \end{pmatrix} + \begin{pmatrix} r_t^i\cos(\phi_t^i + \bar{\mu}_{t,\theta}) \\ r_t^i\sin(\phi_t^i + \bar{\mu}_{t,\theta}) \\ 0 \end{pmatrix}$$

endif

$$\delta = \begin{pmatrix} \delta_x \\ \delta_y \end{pmatrix} = \begin{pmatrix} \bar{\mu}_{j,x} - \bar{\mu}_{t,x} \\ \bar{\mu}_{j,y} - \bar{\mu}_{t,y} \end{pmatrix}$$

$q = \delta^T\delta$

$$\hat{z}_t^i = \begin{pmatrix} \sqrt{q} \\ \operatorname{atan2}(\delta_y, \delta_x) - \bar{\mu}_{t,\theta} \\ \bar{\mu}_{j,s} \end{pmatrix}$$

$$F_{x,j} = \begin{pmatrix} 1 & 0 & 0 & 0\cdots0 & 0 & 0 & 0 & 0\cdots0 \\ 0 & 1 & 0 & 0\cdots0 & 0 & 0 & 0 & 0\cdots0 \\ 0 & 0 & 1 & 0\cdots0 & 0 & 0 & 0 & 0\cdots0 \\ 0 & 0 & 0 & 0\cdots0 & 1 & 0 & 0 & 0\cdots0 \\ 0 & 0 & 0 & 0\cdots0 & 0 & 1 & 0 & 0\cdots0 \\ 0 & 0 & 0 & \underbrace{0\cdots0}_{3j-3} & 0 & 0 & 1 & \underbrace{0\cdots0}_{3N-3j} \end{pmatrix}$$

$$H_t^i = \frac{1}{q}\begin{pmatrix} -\sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 & +\sqrt{q}\delta_x & \sqrt{q}\delta_y & 0 \\ \delta_y & -\delta_x & -q & -\delta_y & +\delta_x & 0 \\ 0 & 0 & 0 & 0 & 0 & q \end{pmatrix} F_{x,j}$$

$K_t^i = \bar{\Sigma}_t\,H_t^{iT}(H_t^i\,\bar{\Sigma}_t\,H_t^{iT} + Q_t)^{-1}$

$\bar{\mu}_t = \bar{\mu}_t + K_t^i(z_t^i - \hat{z}_t^i)$

$\bar{\Sigma}_t = (I - K_t^i\,H_t^i)\,\bar{\Sigma}_t$

endfor

$\mu_t = \bar{\mu}_t$

$\Sigma_t = \bar{\Sigma}_t$

return μ_t, Σ_t

Table 2: The EKF algorithm for the SLAM problem (with known correspondences) [3]

As a result, the simulation is set up as the following. Process noise (R) added to the control inputs to simulate noise experienced in the real world. A set of truth values are computed from the raw control values.

$$R = \begin{bmatrix} \sigma_v \\ \sigma_w \end{bmatrix} \tag{10}$$

$$X_{true} = FX + B(U) \tag{11}$$

The implementation of the motion model prediction code is shown in `motion_ model`. The observation function shows how the simulation uses the process noise $\mathrm{R_{sim}}$ to the find the ground truth.

2. Predicted Covariance
Added the state covariance to the the current uncertainty of the EKF. At each time step, the uncertainty in the system grows by the covariance of the pose, Cx.

$$P = G^T PG + Cx \tag{12}$$

Here main point to note is only growing with respect to the pose, not to the landmarks. - Update the belief in landmark positions based on the estimated state and measurements.
In the update phase, the observations of nearby landmarks are used to correct the location estimate. For every landmark observed, it is associated to a particular landmark in the known map. If no landmark exists in the position surrounding the landmark, it is taken as a NEW landmark. The distance threshold for how far a landmark must be from the next known landmark before its considered to be a new landmark is set by 'DIST_TH'. With an observation associated to the appropriate landmark, the difference can be calculated.

$$y = z_t - h(X) \tag{13}$$

With the innovation calculated, the question becomes which to trust more - the observations or the predictions? To determine this, we calculate the Kalman Gain - a percent of how much of the difference to add to the prediction based on the uncertainty in the predict step and the update step.

$$K = \bar{P}_t H_t^T (H_t \bar{P}_t H_t^T + Q_t)^{-1} \tag{14}$$

In these equations, H is the jacobian of the measurement function. The multiplications by H^T and H represent the application of the delta to the measurement covariance. Intuitively, this equation is applying the following from the single variate Kalman equation but in the multivariate form, i.e. finding the ratio of the uncertianty of the process compared the measurment.

$$K = \frac{\bar{P}_t}{\bar{P}_t + Q_t} \tag{15}$$

If $Q_t << \bar{P}_t$, (i.e. the measurement covariance is low relative to the current estimate), then the Kalman gain will be 1. This results in adding all of the innovation to the estimate – and therefore completely believing the measurement. However, if $Q_t >> \bar{P}_t$ then the Kalman gain will go to 0, signaling a high trust in the process and little trust in the measurement. The update is captured in the following.

$$x_{\text{Update}} = x_{\text{Est}} + (K * y) \tag{16}$$

The covariance must also be updated as well to account for the changing uncertainty. Localization error can be found in terms of variance values for the robot at loop closure.

$$P_t = (I - K_t H_t)\bar{P}_t \tag{17}$$

Observation:
The observation step described here is outside the main EKF SLAM process and is primarily used as a method of driving the simulation. The observations funciton is in charge of calcualting how the poses of the robots change and accumulate error over time, and the theoretical measurements that are expected as a result of each measurement. Observations are based on the TRUE position of the robot.

3 Simulation and Results

3.1 Simulation

The table 2 summarizes the parameter values used for which robot achieves loop closure.

Simulation Parameters	**Value**
No of Landmarks (*static predefined*)	100
Total Simulation Time	150 Sec
Maximum Observation Range	10 m
Threshold distance for data association	2 m
Linear Velocity	10.0 m/s
Yaw Rate	0.6 rad/s

Table 3: Simulation Parameters

The 100 landmarks are created randomly, distributed along -40 to 40 and 0 to 30 meters in x,y directions respectively. Mobile robot starts from (0,0) and moves, Whenever it observes a landmark in the terminal outputs *"New Landmark Found"*

3.2 Results

In EKF-SLAM simulation following conventions are used,

1. Black stars: True Landmarks
2. Blue stars: Estimates of landmark positions
3. Black pointer: True Position/Trajectory
4. Blue pointer: EKF SLAM Position/Trajectory

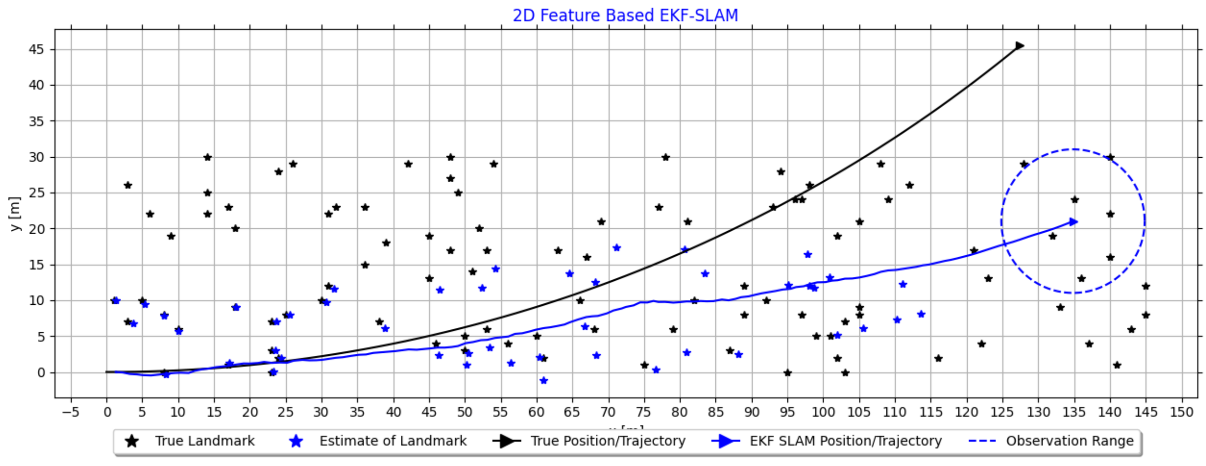

Figure 1: EKF-SLAM without Loop Closure I

The more away robot moving from starting point uncertainty grows. When it observes landmarks, uncertainty increases in landmarks too. Because robot have its own uncertainty and it observes landmarks with that uncertainty, so robot is performing SLAM but not in a useful manner. It is noted that over time uncertainties increases without loop closure(recognizing an already mapped area). It can be seen that blue stars and black stars are not aligned. (*see figure 1, 2*).

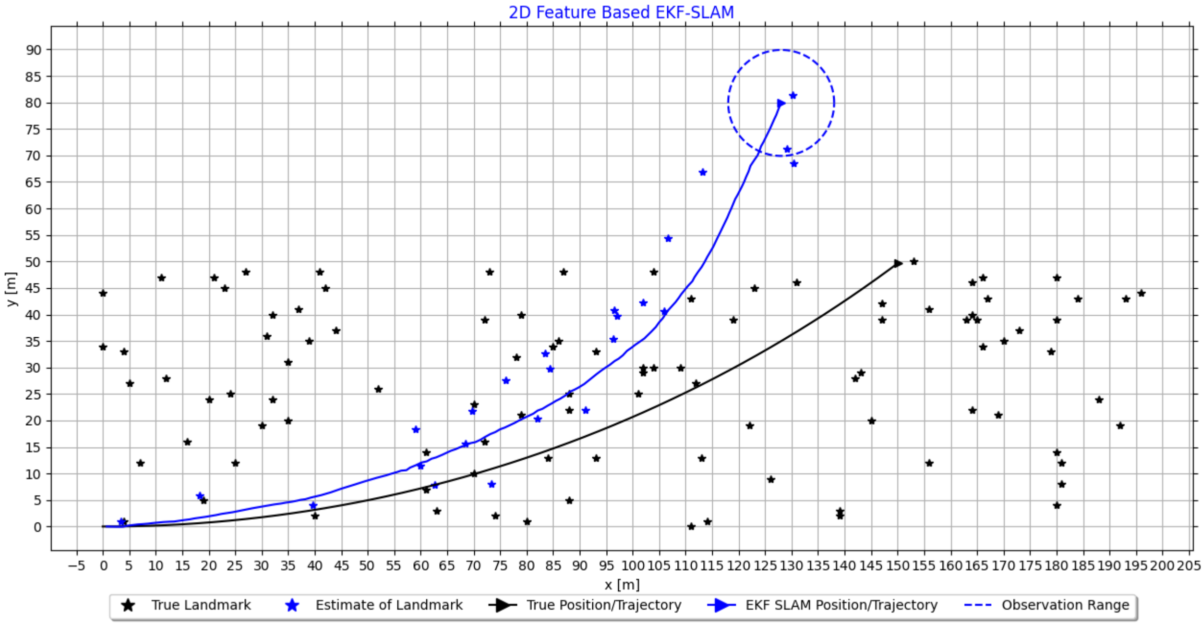

Figure 2: EKF-SLAM without Loop Closure II

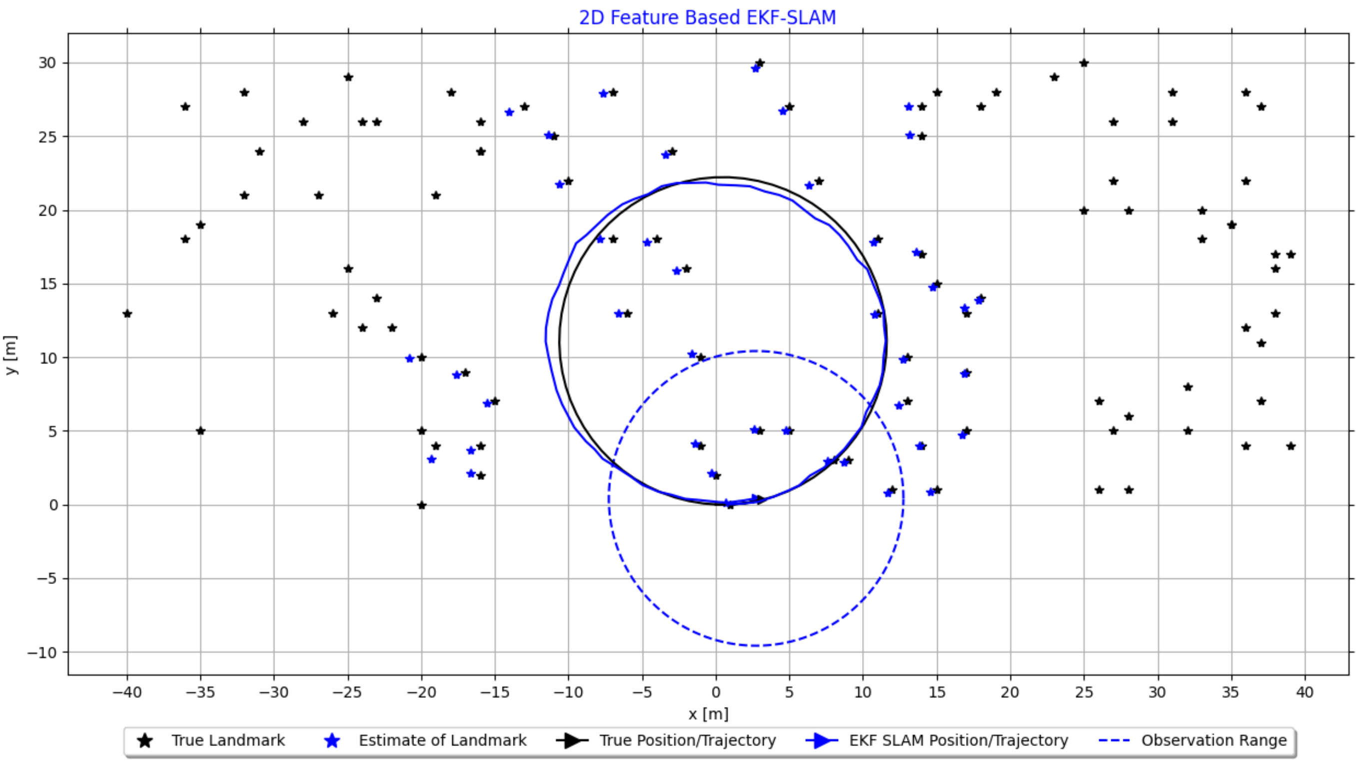

Figure 3: EKF-SLAM with Loop Closure

By achieving loop closure, `uncertainties reduced suddenly.` Noted that Black stars and Blue stars aligning with each other with small error. (*see figure 2*) With loop closure robot can correct its own uncertainty and correction can be propagated to all the landmarks in the maps. After loop closure co-variance matrix, uncertainty in robot and map elements (Diagonal elements) reduces, then it propagates to off diagonal elements.

Bibliography

[1] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, José Neira, Ian Reid, and John J. Leonard. Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Transactions on Robotics*, 32(6):1309–1332, Dec 2016.

[2] Atsushi Sakai, Daniel Ingram, Joseph Dinius, Karan Chawla, Antonin Raffin, and Alexis Paques. Pythonrobotics: a python code collection of robotics algorithms, 2018.

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

Appendix

.1 Python Code

Function	**Objective**
ekf_slam	Performs an iteration of EKF SLAM from the available information.
calc_input	Calculates the control input.
motion_model	Computes the motion model based on current state and input function.
observation	Calcualtes how the poses of the robots change and accumulate error over time.
calc_n_lm	Calculates the number of landmarks currently tracked in the state.
jacob_motion	Calculates the jacobian of motion model
calc_innovation	Calculates the innovation based on expected position and landmark position.
jacobh	Calculates the jacobian of the measurement function.

Table 4: Implemeted Python Functions

```

/*##########Script Information##############################
  # Purpose: 2D Feature based EKF - SLAM
  # Created: 28-10-2021
  # Author        : Atsushi Sakai
  # Modified by   : Rumesh
  ###########################################################*/

import math
import matplotlib.pyplot as plt
import matplotlib.lines as mlines
import numpy as np

# Simulation parameter
Q_sim = np.diag([0.2, np.deg2rad(1.0)]) ** 2  # Sensor Noise
R_sim = np.diag([1.0, np.deg2rad(10.0)]) ** 2 # Process Noise

DT = 0.1              # time step [s]
SIMT = 150.0          # simulation time [s]
MAX_RANGE = 10.0      # maximum observation range
DIST_TH = 2.0         # Threshold distance for data association.
STATE_SIZE = 3        # State size [x,y,yaw]
LM_SIZE = 2           # Landmark state size (2D SLAM)[x,y]

# EKF state covariance
Cx = np.diag([0.5, 0.5, np.deg2rad(30.0)]) ** 2


# Table 2: The EKF algorithm for the SLAM problem
def ekf_slam(xEst, PEst, u, z):
    # Predict
    S = STATE_SIZE
    G, Fx = jacob_motion(xEst[0:S], u)
    xEst[0:S] = motion_model(xEst[0:S], u)
    PEst[0:S, 0:S] = G.T @ PEst[0:S, 0:S] @ G + Fx.T @ Cx @ Fx
    initP = np.eye(2)

    # Update
    for iz in range(len(z[:, 0])):  # for each observation
        min_id = search_correspond_landmark_id(xEst, PEst, z[iz, 0:2])

        nLM = calc_n_lm(xEst)
        if min_id == nLM:
            # print("New Landmark Found")
            # Extend state and covariance matrix
            xAug = np.vstack((xEst, calc_landmark_position(xEst, z[iz, :])))
            PAug = np.vstack((np.hstack((PEst, np.zeros((len(xEst), LM_SIZE)))),
                              np.hstack((np.zeros((LM_SIZE, len(xEst))), initP))))
            xEst = xAug
            PEst = PAug
        lm = get_landmark_position_from_state(xEst, min_id)
        y, S, H = calc_innovation(lm, xEst, PEst, z[iz, 0:2], min_id)

        K = (PEst @ H.T) @ np.linalg.inv(S)
        xEst = xEst + (K @ y)
        PEst = (np.eye(len(xEst)) - (K @ H)) @ PEst

    xEst[2] = pi_2_pi(xEst[2])
    return xEst, PEst

# Control Input to the mobile robot (Eq 7)
def calc_input():
    v = 10.0  # [m/s]
    #yaw_rate = 0.04  # [rad/s] For move robot without loop closure
    yaw_rate = 0.9  # [rad/s]
    u = np.array([[v, yaw_rate]]).T
    return u


# Motion model (Eq 5 - 9)
def motion_model(x, u):
    F = np.array([[1.0, 0, 0],
```

```
                    [0, 1.0, 0],
                    [0, 0, 1.0]])

    B = np.array([[DT * math.cos(x[2, 0]), 0],
                    [DT * math.sin(x[2, 0]), 0],
                    [0.0, DT]])

    x = (F @ x) + (B @ u)
    return x



# Observation model
def observation(xTrue, u, RFID):
    xTrue = motion_model(xTrue, u)

    # add noise to gps x-y
    z = np.zeros((0, 3))

    for i in range(len(RFID[:, 0])):

        dx = RFID[i, 0] - xTrue[0, 0]
        dy = RFID[i, 1] - xTrue[1, 0]
        d = math.hypot(dx, dy)
        angle = pi_2_pi(math.atan2(dy, dx) - xTrue[2, 0])
        if d <= MAX_RANGE:
            # Adding sensor noise
            dn = d + np.random.randn() * Q_sim[0, 0] ** 0.5
            angle_n = angle + np.random.randn() * Q_sim[1, 1] ** 0.5
            zi = np.array([dn, angle_n, i])
            z = np.vstack((z, zi))

    # Adding process noise
    ud = np.array([[
        u[0, 0] + np.random.randn() * R_sim[0, 0] ** 0.5,
        u[1, 0] + np.random.randn() * R_sim[1, 1] ** 0.5]]).T

    return xTrue, z, ud


def calc_n_lm(x):
   n = int((len(x) - STATE_SIZE) / LM_SIZE)
   return n


def jacob_motion(x, u):
    Fx = np.hstack((np.eye(STATE_SIZE), np.zeros(
        (STATE_SIZE, LM_SIZE * calc_n_lm(x)))))

    jF = np.array([[0.0, 0.0, -DT * u[0, 0] * math.sin(x[2, 0])],
                     [0.0, 0.0, DT * u[0, 0] * math.cos(x[2, 0])],
                     [0.0, 0.0, 0.0]], dtype=float)

    G = np.eye(STATE_SIZE) + Fx.T @ jF @ Fx
    return G, Fx,


def calc_landmark_position(x, z):
    zp = np.zeros((2, 1))

    zp[0, 0] = x[0, 0] + z[0] * math.cos(x[2, 0] + z[1])
    zp[1, 0] = x[1, 0] + z[0] * math.sin(x[2, 0] + z[1])

    return zp


def get_landmark_position_from_state(x, ind):
    lm = x[STATE_SIZE + LM_SIZE * ind: STATE_SIZE + LM_SIZE * (ind + 1), :]
    return lm


def search_correspond_landmark_id(xAug, PAug, zi):
    """
```

```python
    Landmark association with Threshold distance
    """
    nLM = calc_n_lm(xAug)
    min_dist = []

    for i in range(nLM):
        lm = get_landmark_position_from_state(xAug, i)
        y, S, H = calc_innovation(lm, xAug, PAug, zi, i)
        min_dist.append(y.T @ np.linalg.inv(S) @ y)

    min_dist.append(DIST_TH)  # new landmark
    min_id = min_dist.index(min(min_dist))

    return min_id


def calc_innovation(lm, xEst, PEst, z, LMid):
    delta = lm - xEst[0:2]
    q = (delta.T @ delta)[0, 0]
    z_angle = math.atan2(delta[1, 0], delta[0, 0]) - xEst[2, 0]
    zp = np.array([[math.sqrt(q), pi_2_pi(z_angle)]])
    y = (z - zp).T
    y[1] = pi_2_pi(y[1])
    H = jacob_h(q, delta, xEst, LMid + 1)
    S = H @ PEst @ H.T + Cx[0:2, 0:2]
    return y, S, H


def jacob_h(q, delta, x, i):
    sq = math.sqrt(q)
    G = np.array([[-sq * delta[0, 0], - sq * delta[1, 0], 0, sq * delta[0, 0], sq * delta[1, 0]],
                  [delta[1, 0], - delta[0, 0], - q, - delta[1, 0], delta[0, 0]]])

    G = G / q
    nLM = calc_n_lm(x)
    F1 = np.hstack((np.eye(3), np.zeros((3, 2 * nLM))))
    F2 = np.hstack((np.zeros((2, 3)), np.zeros((2, 2 * (i - 1))),
                    np.eye(2), np.zeros((2, 2 * nLM - 2 * i))))

    F = np.vstack((F1, F2))
    H = G @ F
    return H


def pi_2_pi(angle):
    return (angle + math.pi) % (2 * math.pi) - math.pi

show_animation = True

def main():
    print(__file__ + " EKF-SLAM started..")

    time = 0.0
    # Predefined landmark positions [x, y]
    # Generated random landmark distributed in x ==> -40,40, y ==> 0,30
    #   for i in range(100):
    #   x = [random.randint(-40,40) for _ in range(1)]
    #   y = [random.randint(0,30) for _ in range(1)]
    #   z = [x[0], y[0]]


    RFID = np.array([ [32, 8] , [39, 4] , [-31, 24] , [-16, 26] , [-40, 13] , [5, 27] , [-2, 16] ,
                    [-36, 18] , [38, 16] , [18, 14] , [35, 19] , [36, 22] , [23, 29] , [37, 11] ,
                    [-24, 12] , [36, 4] , [15, 28] , [13, 10] , [13, 7] , [-10, 22] , [-20, 5] ,
                    [14, 4] , [11, 13] , [27, 22] , [-7, 18] , [1, 0] , [-13, 27] , [-16, 24] ,
                    [38, 13] , [-3, 24] , [-35, 5] , [37, 27] , [28, 20] , [-20, 0] , [14, 25] ,
                    [39, 17] , [7, 22] , [-1, 10] , [-17, 9] , [14, 17] , [27, 5] , [-28, 26]
```

```
                      ,
                      [36, 28] , [31, 28] , [-16, 24] , [25, 30] , [17, 9] , [8, 3] , [-23, 26] ,
                      [-6, 13] , [-4, 18] , [3, 5] , [35, 19] , [-20, 10] , [26, 1] , [19, 28] ,
                      [-27, 21] , [15, 1] , [-19, 4] , [-25, 29] , [38, 17] , [-18, 28] , [36, 12] ,
                      [-35, 19] , [15, 15] , [-7, 28] , [32, 5] , [18, 27] , [-26, 13] , [-32, 21] ,
                      [11, 18] , [-11, 25] , [-19, 21] , [28, 1] , [-22, 12] , [17, 13] , [28, 6] ,
                      [-36, 27] , [33, 20] , [-24, 26] , [12, 1] , [31, 26] , [-25, 16] , [26, 7] ,
                      [14, 27] , [17, 5] , [5, 5] , [9, 3] , [-16, 2] , [-15, 7] , [3, 30] , [27, 26] ,
                      [0, 2] , [-16, 4] , [25, 20] , [-23, 14] , [-32, 28] , [37, 7] , [-1, 4] , [33, 18] ])

    # RFID = np.array([ [180, 39] , [32, 24] , [73, 48] , [63, 3] , [139, 3] , [78, 32] , [93, 13] , [0, 44] , [164, 22] , [188, 24] , [131, 46] , [180, 47] , [30, 19] , [180, 4] , [196, 44] , [112, 27] , [169, 21] , [19, 5] , [156, 12] , [164, 40] , [179, 33] , [170, 35] , [5, 27] , [87, 48] , [37, 41] , [143, 29] , [181, 8] , [31, 36] , [24, 25] , [184, 43] , [166, 34] , [72, 16] , [180, 14] , [156, 41] , [20, 24] , [74, 2] , [23, 45] , [104, 30] , [102, 29] , [102, 30] , [114, 1] , [35, 20] , [0, 34] , [109, 30] , [80, 1] , [7, 12] , [122, 19] , [113, 13] , [167, 43] , [32, 40] , [41, 48] , [84, 13] , [147, 42] , [147, 39] , [111, 0] , [79, 21] , [21, 47] , [16, 16] , [44, 37] , [61, 7] , [35, 31] , [39, 35] , [165, 39] , [163, 39] , [61, 14] , [119, 39] , [52, 26] , [4, 1] , [142, 28] , [153, 50] , [101, 25] , [181, 12] , [192, 19] , [12, 28] , [40, 2] , [88, 5] , [72, 39] , [79, 40] , [70, 10] , [123, 45] , [104, 48] , [70, 23] , [193, 43] , [111, 43] , [86, 35] , [42, 45] , [164, 46] , [27, 48] , [166, 47] , [145, 20] , [88, 25] , [139, 2] , [93, 33] , [85, 34] , [11, 47] , [126, 9] , [88, 22] , [173, 37] , [25, 12] , [4, 33] ])

    # State Vector [x y yaw v]
    # Simulation starts from (0,0) Coordinate
    xEst = np.zeros((STATE_SIZE, 1))
    xTrue = np.zeros((STATE_SIZE, 1))
    PEst = np.eye(STATE_SIZE)

    # history
    hxEst = xEst
    hxTrue = xTrue

    while SIMT >= time:
        time += DT
        u = calc_input()

        xTrue, z, ud = observation(xTrue, u, RFID)

        xEst, PEst = ekf_slam(xEst, PEst, ud, z)
        x_state = xEst[0:STATE_SIZE]

        # store data history
        hxEst = np.hstack((hxEst, x_state))
        hxTrue = np.hstack((hxTrue, xTrue))
        print(PEst)

        if show_animation:  # pragma: no cover

            plt.cla()
            # for stopping simulation with the esc key.
            plt.gcf().canvas.mpl_connect(
                'key_release_event',
                lambda event: [exit(0) if event.key == 'escape' else None])


            plt.plot(RFID[:, 0], RFID[:, 1], "*k")
            plt.plot(xEst[0], xEst[1], ">b")
            plt.plot(xTrue[0], xTrue[1], ">k")
            # Observation range circle plot
            theta = np.arange(0, 2*np.pi, 0.01)
            circlex =  (xEst[0]) + 10 * np.cos(theta)
            circley =  (xEst[1]) + 10 * np.sin(theta)
            plt.plot(circlex,circley, color='b',linestyle='--')
```

```python


            plt.title('2D Feature Based EKF-SLAM',color="blue")
            mng = plt.get_current_fig_manager()
            mng.window.state('zoomed')

            black_star = mlines.Line2D([], [], color='black', marker='*', linestyle='None',
                          markersize=10, label='True Landmark')
            green_x = mlines.Line2D([], [], color='blue', marker='*', linestyle='None',
                          markersize=10, label='Estimate of Landmark')
            red_marker = mlines.Line2D([], [], color='blue', marker='>', linestyle='-',
                          markersize=10, label='EKF SLAM Position/Trajectory')
            blue_marker = mlines.Line2D([], [], color='black', marker='>', linestyle='-',
                          markersize=10, label='True Position/Trajectory')
            blue_marker1 = mlines.Line2D([], [], color='blue', linestyle='--',
                          markersize=10, label='Observation Range')
            plt.legend(handles=[black_star, green_x, blue_marker, red_marker,blue_marker1],loc
                ='upper center',
                                bbox_to_anchor=(0.5, -0.07), fancybox=True, shadow=True, ncol
                                    =5)



            # plot landmark
            for i in range(calc_n_lm(xEst)):
                plt.plot(xEst[STATE_SIZE + i * 2],
                         xEst[STATE_SIZE + i * 2 + 1], "*b")
            plt.plot(hxTrue[0, :],
                     hxTrue[1, :], "-k")
            plt.plot(hxEst[0, :],
                     hxEst[1, :], "-b")
            ax = plt.gca()
            plt.gca().xaxis.set_major_locator(plt.MultipleLocator(5))
            plt.gca().yaxis.set_major_locator(plt.MultipleLocator(5))
            ax.set_aspect(1)
            plt.axis([-40, 40, -10, 40])
            plt.xlabel('x [m]')
            plt.ylabel('y [m]')
            plt.grid(True)
            plt.pause(0.1)


if __name__ == '__main__':
    main()
```