



IP PARIS

INSTITUT POLYTECHNIQUE DE PARIS
ENSTA PARIS

CSC_5RO12_TA, Navegation Robotique

TP4, Extented Kalman Filter SLAM

by

Guilherme NUNES TROFINO

supervised by
David FILLIAT
Goran FREHSE

Confidentiality Notice
Non-confidential and publishable report

ROBOTIQUE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET COMMUNICATION

Paris, FR
13 novembre 2024

Table des matières

1 Question 1	2
1.1 Structure du Code	2
1.2 Execution Initiale	3
1.2.1 Utilization	3
1.3 Scenario 1	4
1.4 Scenario 2	4
1.5 Scenario 3	5
2 Question 2	6
2.1 Scenario 1	6
2.2 Scenario 2	6
2.3 Scenario 3	7
3 Question 3	8
4 Question 4	9
5 Appendix	10

1. Question 1

Remarque. Des modifications ont été faites par rapport à l'algorithme original, notamment avec la création de différentes classes et la définition d'une fonction `main`.

1.1. Structure du Code

L'algorithme utilisé dans ce projet suit la structure suivante :

1. `EKF_SLAM` : Extended Kalman Filter Simultaneous Localization and Mapping sous forme de dataclass, qui contient les méthodes suivantes :

- (a) `F()` ;
- (b) `G()` ;
- (c) `H()` ;
- (d) `calc_innovation()` ;
- (e) `compute_iteration()` ;
- (f) `extended_jacobians()` ;
- (g) `search_landmarks_id()` ;

Remarque. Des commentaires et de la documentation ont été ajoutés à l'algorithme pour faciliter sa compréhension et ne seront pas répétés ici.

Remarque. Entre chaque exécution, seule une variable a été variée tandis que les autres sont restées inchangées, garantissant ainsi que l'analyse se concentre uniquement sur la variable en question.

Remarque. L'application de la fonction `Utils.convert_angle()` a été appliquée à tous les angles calculés dans l'algorithme pour garantir que les résultats affichés sur le graphique soient contenus dans $[-\pi, +\pi]$.

Remarque. Un vecteur $\tilde{\mathbf{a}}$ contient des données bruitées.

Remarque. Un vecteur $\hat{\mathbf{a}}$ est une prédition.

Remarque. Lors de cette exécution, chaque fois que le robot passait à moins d'une certaine distance, une fermeture de boucle était considérée, et une ligne jaune était ajoutée au moment où cela se produisait pour représenter cette fermeture de boucle.

1.2. Execution Initiale

Après l'implementation des fonctions et variables qui manque au code source, quelques modifications sur le graphique ont été fait et le résultat initiale, utilisé comme benchmark, est donnée par l'image ci-dessous :

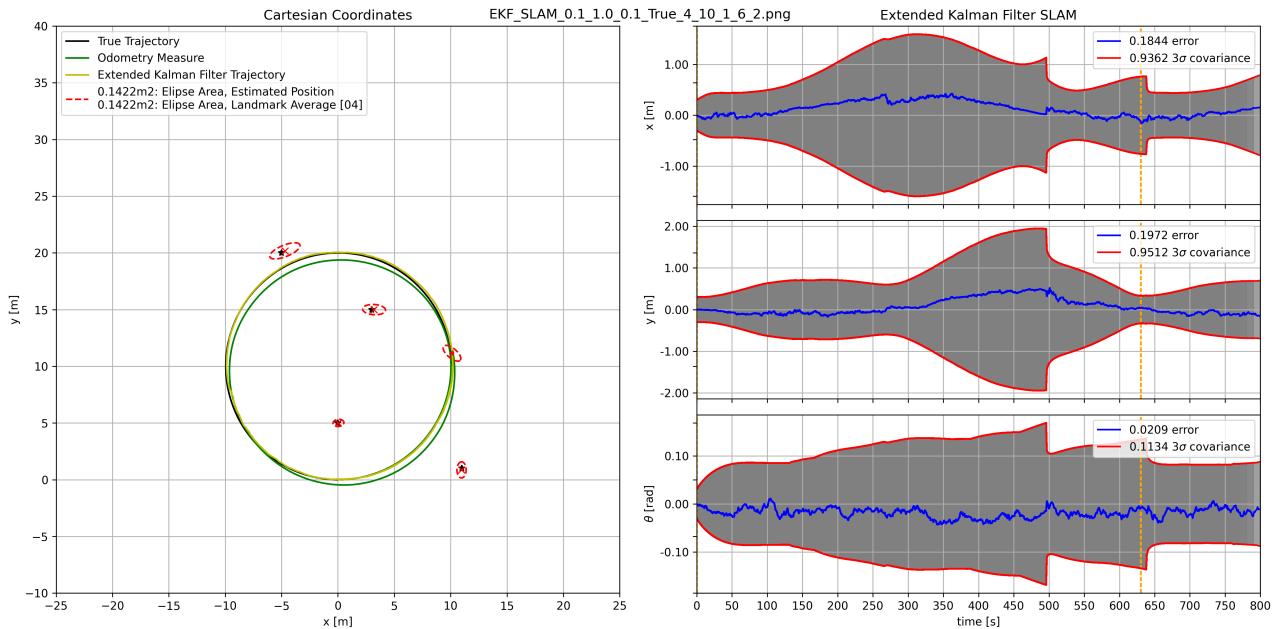


FIGURE 1.1 : Résultats Initiales

Remarque. Dans ce projet, chaque image aura sur son titre les informations sur l'exécution : EKF_SLAM_i_j_k_l_m_n_o_p_q.png où :

1. i : dt : intervalle entre deux prédictions consécutives en secondes ;
2. j : v : vitesse tangentielle de mouvement ;
3. k : w : vitesse angulaire du mouvement ;
4. l : landmarks_know : les coordonnées des repères sont-elles connues ?
5. m : landmarks_count : la quantité de repères dans le scénario ;
6. n : observation_range : la distance d'observation ;
7. o : P_constant : constante de la matrice de covariance d'état \mathbf{P}_k ;
8. p : Q_constant : constante de la matrice de covariance du bruit du processus \mathbf{Q}_k ;
9. q : R_constant : constante de la matrice de covariance du bruit de mesure \mathbf{R}_k ;

1.2.1. Utilization

Pour répondre aux questions, l'algorithme suivant a été utilisé :

```

1 def execution(...) -> None:
2     ...
3
4     plt.suptitle(file_name)
5     if save_result: plt.savefig(file_path, dpi=300)
6     if show_result: plt.show()
7
8 def main():
9     for scenario in [...]:
10         execution(
11             landmarks=scenario['landmarks'],
12             v=scenario['v'],
13             w=scenario['w'],
14             save=True,
15             show=True
16         )
17
18 if __name__ == "__main__":
19     main()

```

1.3. Scenario 1

Définition 1.1. a short loop and a dense map with many landmarks inside the robot perception radius

Ci-dessous, sont présentées quelques interactions résultant de la configuration de **landmarks** du scenario :

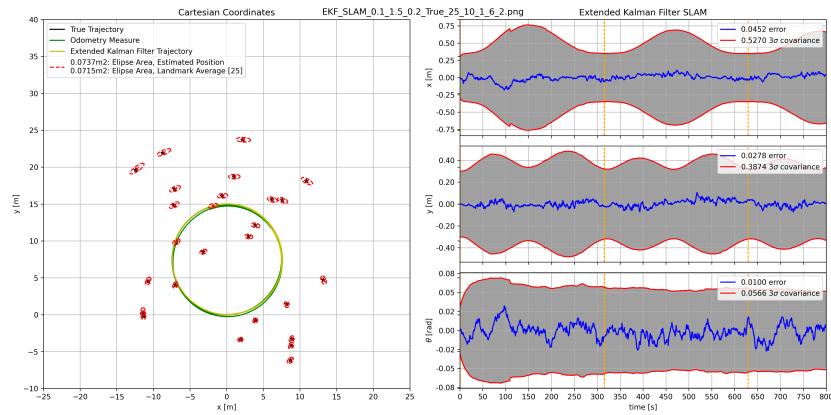


FIGURE 1.2 : Execution Scenario 1, with `landmarks_know = True`

Dans ce cas, il est possible d'observer une oscillation périodique de la variance des coordonnées x et y , due à la rotation de l'ellipse de prédiction. En modifiant son angle, elle change la variation de la coordonnée correspondante. Malgré cela, il est notable que les erreurs restent faibles tout au long du trajet.

On remarque que, juste après les fermetures de boucle, représentées par les lignes jaunes verticales, les erreurs diminuent légèrement pour les coordonnées x et y , mais plus nettement pour l'angle θ . Ce phénomène s'explique par le fait que les fermetures de boucle provoquent une recalibration du filtre.

Cependant, il convient de souligner que chaque nouvel obstacle observé entraîne une mise à jour de l'estimation générale de tous les obstacles, améliorant ainsi l'estimation de la carte.

1.4. Scenario 2

Définition 1.2. a long loop and a dense map with many landmarks all along the loop

Ci-dessous, sont présentées quelques interactions résultant de la configuration de **landmarks** du scenario :

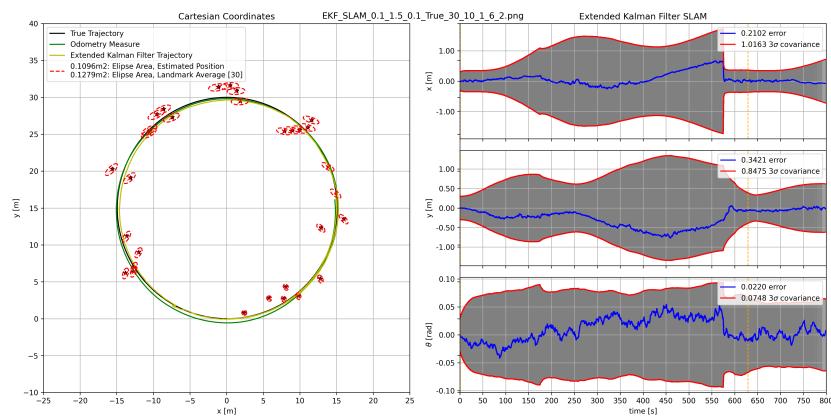


FIGURE 1.3 : Exécution Scenario 2, with `landmarks_know = True`

Dans ce cas, toutes les références ont été placées autour de la boucle, et on observe leur influence sur le résultat. Toutes les coordonnées ont présenté une augmentation significative des erreurs et de la variance, en particulier les coordonnées x et y , qui montraient une croissance importante avant 550 secondes.

Pour les coordonnées x et y , on note une atténuation du phénomène observé dans le scénario précédent. Les variances affichent une croissance plus marquée et une fréquence d'oscillation réduite. Cela pourrait indiquer que les obstacles plus proches de la trajectoire aident à préserver l'estimation.

Entre 550 et 600 secondes, toutes les coordonnées montrent une diminution substantielle des erreurs et des variances. Cela s'explique par une correction de la trajectoire, car une référence a été identifiée, permettant la mise à jour de la carte et de l'estimation des coordonnées du système.

1.5. Scenario 3

Définition 1.3. long loop and a sparse map with only few landmarks near the start position

Ci-dessous, sont présentées quelques interactions résultant de la configuration de **landmarks** du scenario :

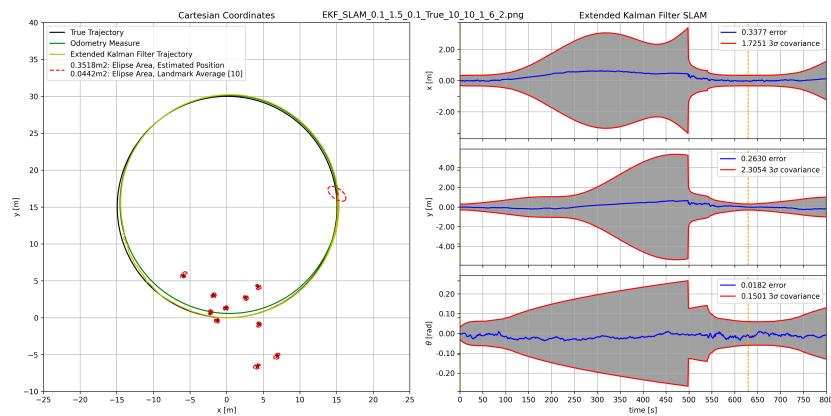


FIGURE 1.4 : Exécution Scenario 3, with `landmarks_know = True`

Dans ce cas, on remarque l'influence de l'absence de références le long de la trajectoire, puisque, après avoir quitté la région initiale, la variance de toutes les coordonnées augmente continuellement. Cela se traduit par une incertitude croissante concernant la position, ce qui s'explique par l'absence de corrections et une dépendance exclusive à l'odométrie.

Aux alentours de 500 secondes, une référence est détectée pour la première fois depuis le départ. Cela permet une correction des estimations du filtre de Kalman, observable par une diminution brusque et significative des erreurs et des covariances à ce moment-là.

2. Question 2

2.1. Scenario 1

Définition 2.1. a short loop and a dense map with many landmarks inside the robot perception radius

Ci-dessous, sont présentées quelques interactions résultant de la configuration de **landmarks** du scenario :

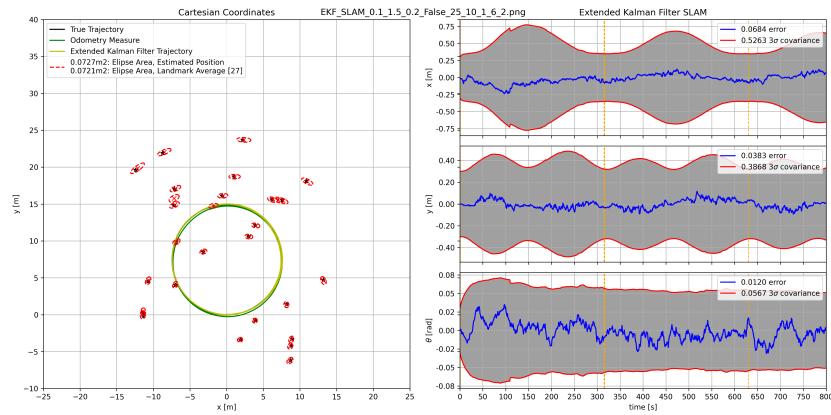


FIGURE 2.1 : Execution Scenario 1, with `landmarks_know = False`

Dans ce cas, on observe que l'utilisation de références estimées, et non plus absolues, entraîne des erreurs et des covariances plus élevées. Cela est attendu, car les références ne garantissent plus leur exactitude, introduisant ainsi davantage d'incertitudes dans la simulation. Néanmoins, le comportement périodique des covariances demeure également dans ce cas.

Il est à noter que, dans ce scénario, il y a plus de références estimées que de références réelles, soit 27 estimées contre 25 véritables. Cela s'explique par le fait que des références proches peuvent entraîner des faux positifs, provoquant une « confusion » du filtre, qui considère qu'il y a une référence là où il n'y en a pas.

Cependant, le résultat global de l'estimation du SLAM par le filtre de Kalman reste d'une qualité acceptable et applicable à des situations pratiques.

2.2. Scenario 2

Définition 2.2. a long loop and a dense map with many landmarks all along the loop

Ci-dessous, sont présentées quelques interactions résultant de la configuration de **landmarks** du scenario :

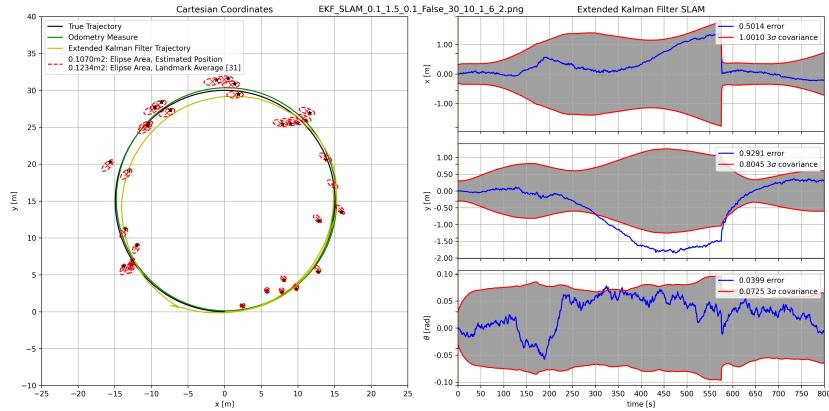


FIGURE 2.2 : Exécution Scenario 2, with `landmarks_know = False`

On remarque que, dans ce cas, le résultat global de l'estimation du SLAM par le filtre de Kalman présente une détérioration considérable, avec l'erreur de la coordonnée y dépassant la limite de 3 covariances. Cela peut être causé par la distribution des références qui, lorsqu'elles se regroupent dans une zone plus restreinte, augmentent la probabilité de "confusion" du filtre. Lors des estimations, on observe une plus grande instabilité, car il devient plus difficile de distinguer des points proches.

Même avec la correction apportée par l'observation d'une référence entre 550 et 600 secondes, le filtre a nécessité davantage d'itérations pour corriger l'erreur accumulée.

2.3. Scenario 3

Définition 2.3. long loop and a sparse map with only few landmarks near the start position

Ci-dessous, sont présentées quelques interactions résultant de la configuration de `landmarks` du scenario :

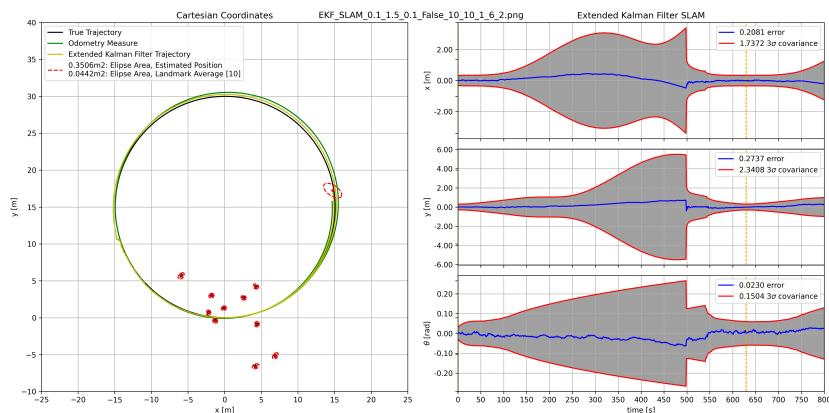


FIGURE 2.3 : Exécution Scenario 3, with `landmarks_know = False`

Dans ce cas, le résultat global de l'estimation du SLAM par le filtre de Kalman reste d'une qualité acceptable et applicable à des situations pratiques, malgré une augmentation générale de l'erreur et de la covariance pour les différentes coordonnées.

Dans les conditions de vitesse tangentielle et angulaire définies, l'odométrie a servi de référence adéquate. Cependant, d'autres valeurs pourraient modifier le résultat final, ce qui pourrait détériorer la performance du filtre..

3. Question 3

Définition 3.1. a short loop and a dense map with many landmarks inside the robot perception radius

Ci-dessous, sont présentées quelques interactions résultant de la configuration de **landmarks** du scenario en variant **Q_constant** à la gauche et **R_constant** à la droite :

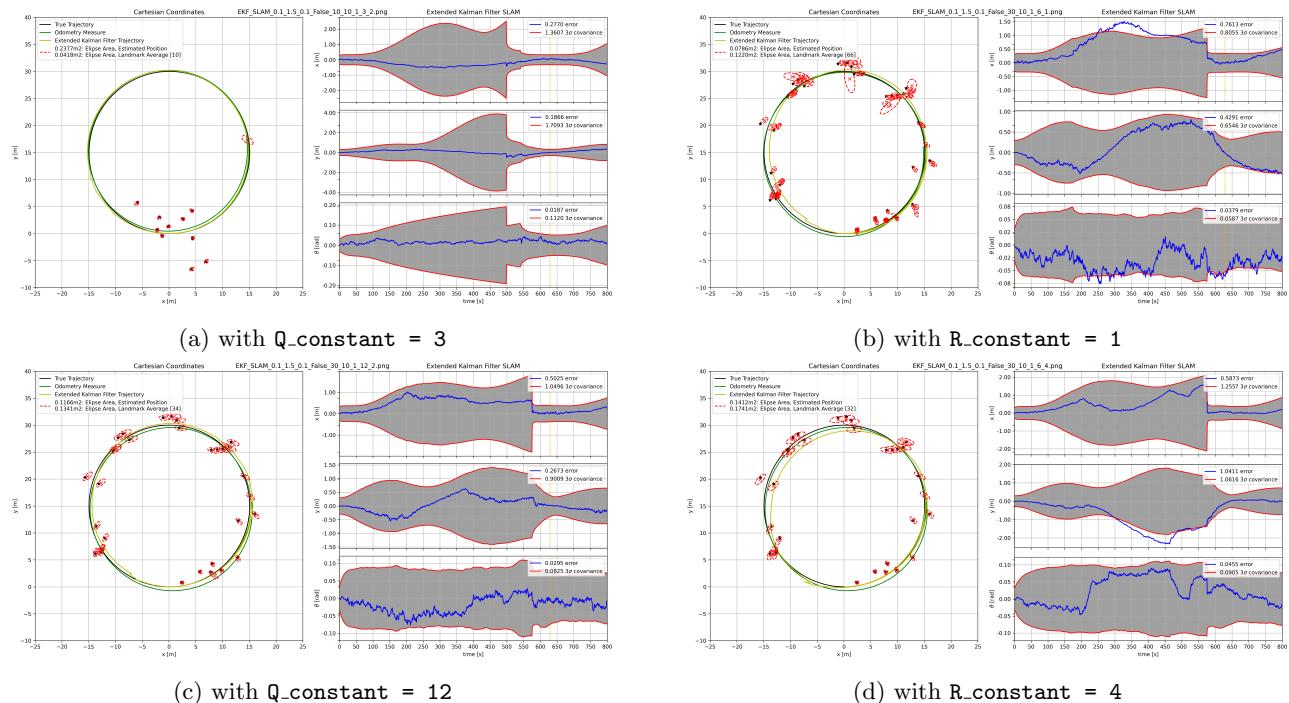


FIGURE 3.1 : Scenario 1, with `landmarks_know = False`

Tout d'abord, le **Q_constant** est analysé, représentant la constante de la matrice de covariance du bruit de processus. On constate que la qualité générale du filtre est inversement proportionnelle à cette constante. Cela signifie que plus la valeur de **Q_constant** est faible, plus le résultat du filtre de Kalman est précis, car les erreurs et les covariances diminuent. Ce résultat est attendu, car plus le bruit de processus est important, plus il est difficile d'estimer correctement le véritable état du système.

Ensuite, on analyse le **R_constant**, la constante de la matrice de covariance du bruit de mesure. On observe que la qualité générale du filtre est plus sensible à cette valeur. Lorsqu'elle est faible (**R_constant = 1**), le filtre manque de variabilité nécessaire pour explorer différents résultats, ce qui entraîne une détérioration visible, avec l'erreur dépassant la limite de 3 covariances.

À l'inverse, lorsqu'elle est élevée (**R_constant = 4**), le filtre présente une variabilité excessive, rendant les résultats trop dispersés pour permettre une estimation cohérente. Cela souligne l'importance d'un réglage précis et minutieux de cette variable.

4. Question 4

Une implémentation préliminaire a été développée dans le code, mais elle n'a pas pu être finalisée. Pendant l'exécution, aucun moyen n'a été trouvé pour effectuer l'élagage (pruning) des différentes covariances au fil de l'exécution. De plus, les estimations des états et des covariances ne correspondaient pas à la référence, rendant le résultat imprécis et peu utile.

Néanmoins, on remarque que cette approche pourrait produire des résultats intéressants dans des situations où les informations sur les références ne sont pas disponibles, comme cela peut être le cas dans une application réelle, bien que cela implique une charge de calcul plus importante.

5. Appendix

```

1 """
2 [CSC_5RO12_TA_TP4] Navegation pour la Robotique, Extended Kalman Filter SLAM
3
4 author: Atsushi Sakai (@Atsushi_twi)
5 modified : Goran Frehse, David Filliat
6 modified : Guilherme Trofino
7 """
8 from dataclasses import dataclass
9 from matplotlib.ticker import FormatStrFormatter
10
11 import math
12 import matplotlib.pyplot as plt
13 import numpy as np
14 import os
15
16 seed = 123456
17 np.random.seed(seed)
18 import os
19 try:
20     os.makedirs("../outputs")
21 except:
22     pass
23
24
25 S_S = 3 # state size: [x, y, omega]
26 L_S = 2 # state size landmarks: [x, y]
27
28
29
30 @dataclass
31 class EKF_SLAM:
32     """Extended Kalman Filter for SLAM methods."""
33
34     def F(x: np.ndarray[float], u: np.ndarray[float], dt: float) -> np.ndarray[float]:
35         """
36             Return motion model state jacobian matrix F(x) as np.ndarray[float].
37
38             Note: jacobian with respect of system state.
39
40             Args:
41                 x (np.ndarray[float]) : system state at instant k-1.
42                 u (np.ndarray[float]) : control input, or odometry measurement, at instant k.
43                 dt (float) : simulation time step in seconds.
44
45             _, _, theta = x[0, 0], x[1, 0], x[2, 0]
46             v, _ = u[0, 0], u[1, 0]
47
48             F = np.array([
49                 [1.0, 0.0, -v * math.sin(theta) * dt],
50                 [0.0, 1.0, +v * math.cos(theta) * dt],
51                 [0.0, 0.0, 1.0]
52             ])
53
54
55             return F
56
57
58     def G(x: np.ndarray[float], u: np.ndarray[float], dt: float) -> np.ndarray[float]:
59         """
60             Return motion model control jacobian matrix G(x) as np.ndarray[float].
61
62             Note: jacobian with respect of noised odometry.
63
64             Args:
65                 x (np.ndarray[float]) : system state at instant k-1.
66                 u (np.ndarray[float]) : control input, or odometry measurement, at instant k.
67                 dt (float) : simulation time step in seconds.
68
69             _, _, theta = x[0, 0], x[1, 0], x[2, 0]
70
71             G = np.array([
72                 [math.cos(theta) * dt, 0.0],
73                 [math.sin(theta) * dt, 0.0],
74                 [0.0, dt]
75             ])
76
77             return G

```

```

78
79     def H(
80         distance: float,
81         relative_position: np.ndarray[float],
82         x: np.ndarray[float],
83         landmark_index: int
84     ) -> np.ndarray[float]:
85         """
86             Return observation model jacobian matrix H(x) for a specific landmark as np.ndarray[float].
87
88         Args:
89             distance (float): squared Euclidean distance between the robot and the landmark.
90             relative_position (np.ndarray[float]): relative position vector [delta_x, delta_y]^T.
91             x (np.ndarray[float]): system state estimation at instant k.
92             landmark_index (int): landmark index in the state vector.
93         """
94         delta_x, delta_y = relative_position[0, 0], relative_position[1, 0]
95
96         sq = math.sqrt(distance)
97         G = np.array([
98             [-sq * delta_x, -sq * delta_y, 0, sq * delta_x, sq * delta_y],
99             [delta_y, -delta_x, -distance, -delta_y, delta_x]
100            ])
101        G = G / distance
102
103        landmark_count = Utils.count_landmarks(x)
104
105        F1 = np.hstack((
106            np.eye(3),
107            np.zeros((3, 2 * landmark_count))
108        ))
109        F2 = np.hstack((
110            np.zeros((2, 3)),
111            np.zeros((2, 2 * landmark_index)),
112            np.eye(2),
113            np.zeros((2, 2 * landmark_count - 2 * (landmark_index + 1)))
114        ))
115
116        F = np.vstack((F1, F2))
117
118        H = G @ F
119
120        return H
121
122
123
124    def calc_innovation(
125        landmark_index,
126        y,
127        x_estimation,
128        P_estimation,
129        R_estimation,
130    ):
131        """
132            Compute innovation vector and related matrices for Kalman gain in EKF-SLAM.
133
134        Args:
135            landmark_index (int): Index of the landmark in the state vector.
136            y (np.ndarray): observed landmark at instant k.
137            x_estimation (np.ndarray[float]): system state estimation at instant k.
138            P_estimation (np.ndarray[float]): system state estimation covariance at instant k.
139            R_estimation (np.ndarray[float]): measure noise covariance matrix at instant k.
140        """
141        # Compute predicted observation from state
142        landmark_position = Utils.get_landmark_position_state(x_estimation, landmark_index)
143        relative_position = landmark_position - x_estimation[0:2]
144
145        distance = (relative_position.T @ relative_position)[0, 0]
146        y_angle = math.atan2(relative_position[1, 0], relative_position[0, 0]) - x_estimation[2, 0]
147
148        y_prediction = np.array([[math.sqrt(distance), Utils.convert_angle(y_angle)]])
149
150        # compute innovation, i.e. diff with real observation
151        innovation = (y - y_prediction).T
152        innovation[1] = Utils.convert_angle(innovation[1])
153
154        # compute matrixes for Kalman Gain
155        H = EKF_SLAM.H(distance, relative_position, x_estimation, landmark_index)
156        S = H @ P_estimation @ H.T + R_estimation
157
158        return innovation, S, H
159
160
161    def compute_iteration(
162        y: np.ndarray[float],

```

```

163     u: np.ndarray[float],
164     x_estimation: np.ndarray[float],
165     P_estimation: np.ndarray[float],
166     Q_estimation: np.ndarray[float],
167     R_estimation: np.ndarray[float],
168     dt: float,
169     landmarks_known: bool,
170     landmarks_true_id: list,
171   ):
172   """
173   Execute Extended Kalman Filter for SLAM prediction and correction.
174
175   Args:
176
177     u (np.ndarray[float]) : control input, or odometry measurement, at instant k-1.
178     x_estimation (np.ndarray[float]) : system state estimation at instant k-1.
179     P_estimation (np.ndarray[float]) : system state estimation covariance at instant k-1.
180     Q_estimation (np.ndarray[float]) : process noise covariance matrix at instant k-1.
181     R_estimation (np.ndarray[float]) : measure noise covariance matrix at instant k-1.
182     dt (float) : simulation time step in seconds.
183
184   """
185   # Kalman States Prediction
186   x_estimation[0:S_S] = Utils.compute_motion(x_estimation[0:S_S], u, dt)
187
188   F = EKF_SLAM.F(x_estimation[0:S_S], u, dt)
189   G = EKF_SLAM.G(x_estimation[0:S_S], u, dt)
190
191   P_estimation[0:S_S, 0:S_S] = F @ P_estimation[0:S_S, 0:S_S] @ F.T + G @ Q_estimation @ G.T
192
193   P_estimation[0:S_S, S_S:] = F @ P_estimation[0:S_S, S_S:]
194   P_estimation[S_S:, 0:S_S] = P_estimation[0:S_S, S_S:].T
195
196   P_estimation = 0.5 * (P_estimation + P_estimation.T)      # symmetry matrix
197
198
199   # Kalman Correction
200   for iy in range(len(y[:, 0])):
201     landmark_count = Utils.count_landmarks(x_estimation)
202
203     if landmarks_known:
204       try:
205         min_id = landmarks_true_id.index(y[iy, 2])
206       except ValueError:
207         min_id = landmark_count
208         landmarks_true_id.append(y[iy, 2])
209     else:
210       min_id = EKF_SLAM.search_landmark_id(
211         y[iy, 0:2], x_estimation, P_estimation, R_estimation
212       )
213
214
215     # Extend map if required
216     if min_id == landmark_count:
217       print("New LM")
218
219     # Extend state matrix
220     x_estimation = np.vstack(
221       x_estimation, Utils.get_landmark_position_absolute(x_estimation, y[iy, :])
222     )
223
224
225     # Extend covariance matrix
226     Jr, Jy = EKF_SLAM.extended_jacobians(x_estimation[0:3], y[iy, :])
227     bottomPart = np.hstack((
228       Jr @ P_estimation[0:3, 0:3], Jr @ P_estimation[0:3, 3:]
229     ))
230     rightPart = bottomPart.T
231
232     P_estimation = np.vstack((
233       np.hstack((P_estimation, rightPart)),
234       np.hstack((bottomPart,
235                 Jr @ P_estimation[0:3, 0:3] @ Jr.T + Jy @ R_estimation @ Jy.T))
236     ))
237
238   else:
239     # Perform Kalman update
240     innovation, S, H = EKF_SLAM.calc_innovation(
241       min_id,
242       y[iy, 0:2],
243       x_estimation,
244       P_estimation,
245       R_estimation,
246     )
247     K = (P_estimation @ H.T) @ np.linalg.inv(S)

```

```

248         x_estimation = x_estimation + (K @ innovation)
249
250         P_estimation = (np.eye(len(x_estimation)) - K @ H) @ P_estimation
251         P_estimation = 0.5 * (P_estimation + P_estimation.T)      # symmetry matrix
252
253         x_estimation[2] = Utils.convert_angle(x_estimation[2])
254
255     return x_estimation, P_estimation
256
257
258 def extended_jacobians(
259     x_estimation: np.ndarray[float], y_estimation: np.ndarray[float]
260 ) -> list[np.ndarray[float], np.ndarray[float]]:
261 """
262     Return extended covariances jacobians matrices.
263
264     Note: Jr is the state extended jacobian and Jy is the command extended jacobian.
265
266     Args:
267         x_estimation (np.ndarray[float]): system state estimation at instant k.
268         y_estimation (np.ndarray): observed landmark estimation at instant k.
269     """
270     _, _, theta = x_estimation[0, 0], x_estimation[1, 0], x_estimation[2, 0]
271     v, w = y_estimation[0], y_estimation[1]
272
273     Jr = np.array([
274         [1.0, 0.0, -v * math.sin(theta + w)],
275         [0.0, 1.0, +v * math.cos(theta + w)]
276     ])
277
278     Jy = np.array([
279         [math.cos(theta + w), -v * math.sin(theta + w)],
280         [math.sin(theta + w), +v * math.cos(theta + w)]
281     ])
282
283
284     return Jr, Jy
285
286
287 def search_landmark_id(
288     yi: np.ndarray[float],
289     x_estimation: np.ndarray[float],
290     P_estimation: np.ndarray[float],
291     R_estimation: np.ndarray[float],
292     mahalanobis_distance: float = 9.0
293 ):
294 """
295     Return landmark id with Mahalanobis distance.
296
297     Args:
298         yi (np.ndarray[float]): :
299         x_estimation (np.ndarray[float]): system state estimation at instant k.
300         P_estimation (np.ndarray[float]): system state estimation covariance at instant k.
301         R_estimation (np.ndarray[float]): measure noise covariance matrix at instant k.
302         mahalanobis_distance (float): mahalanobis threshold distance. Default value is 9.0.
303     """
304     dist_min = []
305
306     for i in range(Utils.count_landmarks(x_estimation)):
307         innovation, S, _ = EKF_SLAM.calc_innovation(
308             i,
309             yi,
310             x_estimation,
311             P_estimation,
312             R_estimation,
313         )
314         dist_min.append(innovation.T @ np.linalg.inv(S) @ innovation)
315
316     dist_min.append(mahalanobis_distance)
317
318     return dist_min.index(min(dist_min))
319
320
321
322 @dataclass
323 class Utils:
324     def compute_motion(
325         x_estimation: np.ndarray[float], u_tilde: np.ndarray[float], dt: float
326     ) -> np.ndarray[float]:
327 """
328     Compute system motion based on motion equation as np.ndarray.
329
330     Args:
331         x_estimation (np.ndarray[float]): system state at instant k.
332         u_tilde (np.ndarray[float]): control input, or odometry measurement, at instant k.

```

```

333     dt (float) : simulation time step in seconds.
334     """
335     assert x_estimation.ndim == 2
336     assert u_tilde.ndim == 2
337
338     x, y, theta = x_estimation[0, 0], x_estimation[1, 0], x_estimation[2, 0]
339     v, omega = u_tilde[0, 0], u_tilde[1, 0]
340
341     x_motion = np.array([
342         [x + v * dt * np.cos(theta)],
343         [y + v * dt * np.sin(theta)],
344         [theta + omega * dt],
345     ])
346     x_motion[2, 0] = Utils.convert_angle(x_motion[2, 0])
347
348     return x_motion
349
350
351 def convert_angle(angle: float) -> float:
352     """
353     Return wrapped radian angle to the range [-pi, pi].
354
355     Args:
356         angle (float): angle in radians.
357     """
358     return (angle + math.pi) % (2 * math.pi) - math.pi
359
360
361 def compute_rms_error(arr: np.ndarray[float]) -> float:
362     """
363     Return RMS error of an np.ndarray as a float.
364
365     Args:
366         arr (np.ndarray) : array to analyze.
367     """
368     return np.sqrt(np.mean(arr**2))
369
370
371 def count_landmarks(x_estimation: np.ndarray[float]) -> int:
372     """
373     Return quantity of landmarks in the a state vector x.
374
375     Args:
376         x_estimation (np.ndarray[float]) : system state at instant k.
377     """
378     return int((len(x_estimation) - S_S) / L_S)
379
380
381 def get_landmark_position_absolute(
382     x_estimation: np.ndarray[float],
383     u_tilde: np.ndarray[float]
384 ) -> np.ndarray[float]:
385     """
386     Return landmark absolute position from robot pose and get_observation.
387
388     Args:
389         x_estimation (np.ndarray[float]) : system state at instant k.
390         u_tilde (np.ndarray[float]) : control input, or odometry measurement, at instant k.
391     """
392     x, y, theta = x_estimation[0, 0], x_estimation[1, 0], x_estimation[2, 0]
393     v, omega = u_tilde[0], u_tilde[1]
394
395     landmark_position = np.zeros((2, 1))
396
397     landmark_position[0, 0] = x + v * math.cos(theta + omega)
398     landmark_position[1, 0] = y + v * math.sin(theta + omega)
399
400     return landmark_position
401
402
403 def get_landmark_position_state(x: np.ndarray[float], index: int) -> np.ndarray[float]:
404     """
405     Return landmark state position from robot pose.
406
407     Args:
408         x (np.ndarray[float]) : system state at instant k.
409         index (int) : landmark index on system state.
410     Extract landmark position from state vector
411     """
412     return x[
413         S_S + L_S * index : S_S + L_S * (index + 1),
414     ]
415
416
417 def is_initial_position(

```

```

418     x_initial: np.ndarray[float],
419     x_estimation: np.ndarray[float],
420     dist_threshold: float = 0.1
421 ) -> bool:
422 """
423     Return if estimated position is initial position as boolean based on a distance threshold.
424
425 Args:
426     x_initial (np.ndarray[float]) : system state estimation at instant 0.
427     x_estimation (np.ndarray[float]) : system state estimation at instant k.
428 """
429 x_0, y_0 = x_initial[0, 0], x_initial[1, 0]
430 x_k, y_k = x_estimation[0, 0], x_estimation[1, 0]
431
432 return np.sqrt((x_0 - x_k)**2 + (y_0 - y_k)**2) < dist_threshold
433
434
435 def plot_covariance_ellipse(
436     x_estimation: np.ndarray[float],
437     P_estimation: np.ndarray[float],
438     line: str,
439     axes,
440     landmarks_data: tuple
441 ) -> None:
442 """
443     Plot motion estimation covariance matrix as an ellipse.
444
445 Args:
446     x_estimation (np.ndarray[float]) : system state estimation at instant k.
447     P_estimation (np.ndarray[float]) : system state estimation covariance at instant k-1.
448     line (str) : plot line style.
449     axes () : matplotlib.plot axis to include plot.
450 """
451 P_xy = P_estimation[0:2, 0:2]
452 eigen_values, eigen_vectors = np.linalg.eig(P_xy)
453
454 if eigen_values[0] >= eigen_values[1]:
455     index_big, index_small = 0, 1
456 else:
457     index_big, index_small = 1, 0
458
459 if eigen_values[index_small] < 0:
460     print('Pb with P_xy :\n', P_xy)
461     exit()
462
463 a = math.sqrt(eigen_values[index_big])
464 b = math.sqrt(eigen_values[index_small])
465 area = math.pi * a * b
466
467 circle = np.arange(0, 2 * math.pi + 0.1, 0.1)
468 x = [3 * a * math.cos(angle) for angle in circle]
469 y = [3 * b * math.sin(angle) for angle in circle]
470
471 angle = math.atan2(eigen_vectors[index_big, 1], eigen_vectors[index_big, 0])
472 rotation = np.array([
473     [+math.cos(angle), +math.sin(angle)],
474     [-math.sin(angle), +math.cos(angle)]
475 ])
476
477 ellipse = rotation @ (np.array([x, y]))
478 px = np.array(ellipse[0, :] + x_estimation[0, 0]).flatten()
479 py = np.array(ellipse[1, :] + x_estimation[1, 0]).flatten()
480
481 if landmarks_data == (0, 0):
482     axes.plot(px, py, line)
483 else:
484     landmarks_label = f'{area:2.4f}m^2: Ellipse Area, Estimated Position\n'
485     landmarks_label += f'{landmarks_data[1]/landmarks_data[0]:2.4f}m^2: Ellipse Area, '
486     landmarks_label += f'Landmark Average [{landmarks_data[0]:0.2d}]'
487
488     axes.plot(
489         px,
490         py,
491         line,
492         label = landmarks_label
493     )
494
495 return area
496
497
498 def generate_landmarks(
499     origin: tuple, r_min: float, r_max: float, n: int
500 ) -> np.ndarray[float]:
501 """
502     Generates random points in polar coordinates.

```

```

503
504     Args:
505         origin (tuple) : (x, y) coordinates of circle's center.
506         r_min (float) : minimum radius of the circle.
507         r_max (float) : maximum radius of the circle.
508         n (int) : number of points to generate.
509         full (bool) : consider full circle? Default is True.
510     """
511     angles = np.random.uniform(0, 2 * np.pi, n)
512     distances = np.random.uniform(r_min, r_max, n)
513
514     x = origin[0] + distances * np.cos(angles)
515     y = origin[1] + distances * np.sin(angles)
516
517     for i, j in zip(x, y):
518         print(f'{i:+2.1f}, {j:+2.1f}],')
519
520     return np.column_stack((x, y))
521
522
523
524 class Simulation:
525     def __init__(
526         self,
527         dt,
528         landmarks,
529         simulation_duration,
530         observation_range,
531         x_true,
532         x_odometry,
533         x_estimation,
534         P_estimation,
535         Q_true,
536         R_true,
537     ) -> None:
538         self.dt = dt
539         self.landmarks = landmarks
540         self.simulation_duration = simulation_duration
541
542         self.observation_range = observation_range
543
544         self.x_true = x_true
545         self.x_odometry = x_odometry
546         self.Q_true = Q_true
547         self.R_true = R_true
548
549         self.history_x_true = x_true
550         self.history_x_odometry = x_odometry
551         self.history_x_estimation = x_estimation
552
553         self.history_x_error = np.abs(x_estimation - x_true)
554         self.history_x_covariance = np.sqrt(
555             np.diag(P_estimation[0:S_S, 0:S_S]).reshape(3,1)
556         )
557
558         self.n_steps = int(np.round(simulation_duration/dt))
559         self.history_time = [0]
560         self.landmarks_true_id = []
561         self.plot_axes = self.plot_init()
562
563
564
565     def get_observation(self, v, w, landmarks):
566         """
567             Generate noisy control and get_observation and update true position and dead reckoning
568         """
569         u_true = self.get_robot_control(v, w)
570
571         # add noise to gps x-y
572         y = np.zeros((0, 3))
573
574         for i in range(len(landmarks[:, 0])):
575             dx = landmarks[i, 0] - self.x_true[0, 0]
576             dy = landmarks[i, 1] - self.x_true[1, 0]
577             angle = Utils.convert_angle(math.atan2(dy, dx) - self.x_true[2, 0])
578
579             d = math.hypot(dx, dy)
580             if d <= self.observation_range:
581                 d_tilde = d + np.random.randn() * self.R_true[0, 0] ** 0.5 # add noise
582                 d_tilde = max(d_tilde, 0)
583                 angle_tilde = angle + np.random.randn() * self.R_true[1, 1] ** 0.5 # add noise
584
585                 yi = np.array([d_tilde, angle_tilde, i])
586                 y = np.vstack((y, yi))
587

```

```

588     # add noise to input
589     u_tilde = np.array([
590         u_true[0, 0] + np.random.randn() * self.Q_true[0, 0] ** 0.5,
591         u_true[1, 0] + np.random.randn() * self.Q_true[1, 1] ** 0.5
592     ]).T
593
594     self.x_odometry = Utils.compute_motion(self.x_odometry, u_tilde, self.dt)
595
596     return y, u_tilde
597
598
599 def get_robot_control(self, v: float = 1.0, w: float = 0.1) -> np.ndarray[float]:
600     """
601     Return robot true control command as np.ndarray.
602
603     Note: by default a circular trajectory is generated.
604
605     Args:
606         v (float) : tangencial velocity in m/s. Default value is 1.0 m/s.
607         w (float) : angular velocity in rad/s. Default value is 0.1 rad/s.
608     """
609     u = np.array([[v, w]]).T
610
611     return u
612
613
614 def plot_init(self) -> list:
615     """Return simulation plot axis."""
616     _, (ax1, ax2) = plt.subplots(1, 2, sharey=True, figsize=(16, 8))
617
618     ax3 = plt.subplot(3, 2, 2)
619     ax4 = plt.subplot(3, 2, 4)
620     ax5 = plt.subplot(3, 2, 6)
621
622     return ax1, ax2, ax3, ax4, ax5
623
624
625 def plot_iteration(
626     self,
627     x_estimation: np.ndarray[float],
628     P_estimation: np.ndarray[float],
629     show_legend: bool = False
630 ) -> None:
631     """
632     Update simulation plot.
633
634     Args:
635         x_estimation (np.ndarray[float]) : system state estimation at instant k.
636         P_estimation (np.ndarray[float]) : system state estimation covariance at instant k.
637         show_legend (bool) : add legend to plot? Default value is False.
638     """
639     # Unfold axis
640     ax1, ax2, ax3, ax4, ax5 = self.plot_axes
641
642
643     # Initialize plot
644     x_max, x_min = +25, -25
645     y_max, y_min = +40, -10
646     x_ticks = [i for i in range(x_min, x_max+1, 5)]
647     y_ticks = [i for i in range(y_min, y_max+1, 5)]
648
649     ax1.cla()
650     ax1.grid(True)
651     ax1.set_title('Cartesian Coordinates')
652     ax1.set_xlabel('x [m]')
653     ax1.set_ylabel('y [m]')
654     ax1.set_xticks(x_ticks)
655     ax1.set_yticks(y_ticks)
656     ax1.axis([x_min, x_max, y_min, y_max])
657
658     ax2.set_xticks([])
659
660     ax3.set_title('Extended Kalman Filter SLAM')
661
662     ax5.set_xlabel('time [s]')
663
664
665     # True trajectory and landmark
666     ax1.plot(self.landmarks[:, 0], self.landmarks[:, 1], '*k')
667     ax1.plot(
668         self.history_x_true[0, :],
669         self.history_x_true[1, :],
670         "-k",
671         label="True Trajectory"
672     )

```

```

673
674     # Estimated odometry trajectory
675     ax1.plot(
676         self.history_x_odometry[0, :],
677         self.history_x_odometry[1, :],
678         "-g",
679         label="Odometry Measure"
680     )
681
682     # Estimated EKF trajectory and pose covariance
683     ax1.plot(
684         self.history_x_estimation[0, :],
685         self.history_x_estimation[1, :],
686         "-y",
687         label="Extended Kalman Filter Trajectory"
688     )
689     ax1.plot(x_estimation[0], x_estimation[1], ".r")
690
691
692     # Covariance ellipses
693     landmarks_area = 0
694     landmark_count = Utils.count_landmarks(x_estimation)
695     for i in range(landmark_count):
696         landmark_i = S_S + i * 2
697         ax1.plot(x_estimation[landmark_i], x_estimation[landmark_i + 1], "xr")
698
699         landmarks_area += Utils.plot_covariance_ellipse(
700             x_estimation[landmark_i : landmark_i + 2],
701             P_estimation[landmark_i : landmark_i + 2, landmark_i : landmark_i + 2],
702             "--r",
703             ax1,
704             (0, 0)
705         )
706
707     Utils.plot_covariance_ellipse(
708         x_estimation[0: S_S],
709         P_estimation[0: S_S, 0: S_S],
710         "--r",
711         ax1,
712         (landmark_count, landmarks_area) if show_legend else (0, 0)
713     )
714     ax1.legend(loc='upper left')
715
716
717     # Errors and covariances plot
718     error_axes = [ax3, ax4, ax5]
719     ylabels = ['x [m]', 'y [m]', r"$\theta$ [rad]"]
720     x_ticks = [i for i in range(0, self.simulation_duration*10+1, 50)]
721
722     for i, error_axis in enumerate(error_axes):
723         # plot errors curves
724         if show_legend:
725             rms_err = Utils.compute_rms_error(1 * self.history_x_error[i, :])
726             rms_cov = Utils.compute_rms_error(3 * self.history_x_covariance[i, :])
727
728             label_err = f'{rms_err:2.4f} error'
729             label_cov = f'{rms_cov:2.4f} 3$\sigma$ covariance'
730
731             error_axis.plot(+1.0 * self.history_x_error[i, :], 'b', label=label_err)
732             error_axis.plot(+3.0 * self.history_x_covariance[i, :], 'r', label=label_cov)
733         else:
734             error_axis.plot(+1.0 * self.history_x_error[i, :], 'b')
735             error_axis.plot(+3.0 * self.history_x_covariance[i, :], 'r')
736
737             error_axis.plot(-3.0 * self.history_x_covariance[i, :], 'r')
738
739             error_axis.fill_between(
740                 self.history_time,
741                 +3.0 * self.history_x_covariance[i, :],
742                 -3.0 * self.history_x_covariance[i, :],
743                 color='gray',
744                 alpha=0.75
745             )
746
747             error_axis.yaxis.set_major_formatter(FormatStrFormatter('%.2f'))
748             error_axis.set_ylabel(ylabels[i])
749             error_axis.set_xlim(0, self.simulation_duration)
750             error_axis.set_xticks(x_ticks)
751
752             if error_axis != ax5:
753                 error_axis.set_xticklabels(['' for _ in x_ticks])
754                 error_axis.grid(True)
755
756             if show_legend: error_axis.legend(loc='upper right')
757

```

```

758     plt.tight_layout()
759     plt.pause(0.0001)
760
761
762     def simulate_world(self, v: float = 1.0, w: float = 0.1) -> None:
763         """
764             Simulate system.
765
766             Args:
767                 v (float) : tangencial velocity in m/s. Default value is 1.0 m/s.
768                 w (float) : angular velocity in rad/s. Default value is 0.1 rad/s.
769             """
770             self.x_true = Utils.compute_motion(
771                 self.x_true, self.get_robot_control(v, w), self.dt
772             )
773             self.x_true[2, 0] = Utils.convert_angle(self.x_true[2, 0])
774
775
776     def update_history(
777         self, k: int, x_estimation: np.ndarray[float], P_estimation: np.ndarray[float]
778     ) -> None:
779         """
780             Update simulation history.
781
782             Args:
783                 k (int) : simulation instant.
784                 x_estimation (np.ndarray[float]) : system state estimation at instant k.
785                 P_estimation (np.ndarray[float]) : system state estimation covariance at instant k.
786             """
787             self.history_x_true = np.hstack((self.history_x_true, self.x_true))
788             self.history_x_odometry = np.hstack((self.history_x_odometry, self.x_odometry))
789             self.history_x_estimation = np.hstack((
790                 self.history_x_estimation, x_estimation[0:S_S]
791             ))
792
793             error = x_estimation[0:S_S] - self.x_true
794             error[2, 0] = Utils.convert_angle(error[2, 0])
795
796             self.history_x_error = np.hstack((self.history_x_error, error))
797             self.history_x_covariance = np.hstack((
798                 self.history_x_covariance, np.sqrt(
799                     np.diag(P_estimation[0:S_S, 0:S_S]).reshape(3,1)
800                 )
801             ))
802             self.history_time.append(10 * k * self.dt)
803
804
805
806     def execution(
807         landmarks: np.ndarray[float],
808         landmarks_known: bool = True,
809         v: float = 1.0,
810         w: float = 0.1,
811         dt: float = 0.1,
812         observation_range: int = 10,
813         P_constant: int = 1,
814         Q_constant: int = 6,
815         R_constant: int = 2,
816         save: bool = False,
817         show: bool = True,
818     ) -> None:
819         """
820             Execute an Particles Filter simulation for SLAM.
821
822             Args:
823                 landmarks (np.ndarray[float]) : array of landmarks coordinates.
824                 landmarks_known (bool) : use landmarks true positions? Default value is True.
825                 dt (int) : prediction interval in seconds. Default value is 1.
826                 observation_range (int) : robot observation range. Default value is 10.
827                 P_constant (int) : state noise covariance matrix P constant. Default value is 1.
828                 Q_constant (int) : process noise covariance matrix Q constant. Default value is 1.
829                 R_constant (int) : measure noise covariance matrix R constant. Default value is 1.
830                 save (bool) : save result? Default value is False.
831                 show (bool) : show result? Default value is True.
832             """
833
834             # Define Kalman covariance errors robot movements
835             P_true = np.diag([0.01, 0.01, 0.0001])
836             Q_true = np.diag([0.1, np.deg2rad(1)]) ** 2      # input noise
837             R_true = np.diag([0.1, np.deg2rad(5)]) ** 2      # measurement noise
838
839             P_estimation = P_constant * P_true
840             Q_estimation = Q_constant * Q_true # estimated input noise
841             R_estimation = R_constant * R_true # estimated measurement noise
842

```

```

843     # Simulation initial conditions
844     simulation_duration = 80      # simulation time [s]
845
846     x_true = np.zeros((S_S, 1))
847     x_initial = x_true
848     x_odometry = x_true
849     x_estimation = x_true
850
851     # Simulation creation
852     simulation = Simulation(
853         dt,
854         landmarks,
855         simulation_duration,
856         observation_range,
857         x_true,
858         x_odometry,
859         x_estimation,
860         P_estimation,
861         Q_true,
862         R_true,
863     )
864
865     landmarks_vote = np.zeros(4)
866
867     # Simulation execution
868     for k in range(1, simulation.n_steps):
869         simulation.simulate_world(v, w)
870         z, u_tilde = simulation.get_observation(v, w, landmarks)
871
872         x_estimation, P_estimation = EKF_SLAM.compute_iteration(
873             z,
874             u_tilde,
875             x_estimation,
876             P_estimation,
877             Q_estimation,
878             R_estimation,
879             simulation.dt,
880             landmarks_known,
881             simulation.landmarks_true_id,
882         )
883
884         simulation.update_history(k, x_estimation, P_estimation)
885
886         # Verify loop closure
887         if Utils.is_initial_position(x_initial, x_estimation):
888             for error_axes in simulation.plot_axes[2:]:
889                 error_axes.axvline(x=k, color='orange', linestyle='--', linewidth=1)
890
891         # Simulation plot
892         if show and (k % 10 == 0):
893             simulation.plot_iteration(x_estimation, P_estimation)
894
895         simulation.plot_iteration(x_estimation, P_estimation, show_legend=True)
896
897
898     # Simulation save / show
899     file_name = f'EKF_SLAM_{dt}_{v}_{w}_'
900     file_name += f'{landmarks_known}_{landmarks.shape[0]}_{observation_range}_'
901     file_name += f'{P_constant}_{Q_constant}_{R_constant}.png'
902
903     file_path = os.path.join(os.path.abspath(os.getcwd()), '../outputs', file_name)
904
905     plt.suptitle(file_name)
906     if save: plt.savefig(file_path, dpi=300)
907     if show: plt.show()
908
909
910
911 def main():
912     # Utils.generate_landmarks((0, 15), 13, 17, 30)
913     # return
914
915     default = {
916         'v': 1.0,
917         'w': 0.1,
918         'landmarks': np.array([
919             [+00.0, +05.0], [+11.0, +01.0], [+03.0, +15.0], [-05.0, +20.0]
920         ])
921     }
922
923     # short loop and a dense map
924     scenario_0 = {
925         'v': 1.5,
926         'w': 0.2,
927         'landmarks': np.array([

```

```

928     [+03.0, +10.6], [+13.2, +04.7], [-00.6, +16.1], [+08.2, +01.4], [-12.4, +19.6],
929     [-08.7, +21.9], [-03.2, +08.5], [+08.7, -06.2], [+10.9, +18.1], [-10.7, +04.5],
930     [-07.2, +14.8], [-06.9, +09.8], [+07.6, +15.5], [+08.9, -03.3], [+03.9, -00.8],
931     [-07.1, +17.0], [+02.2, +23.7], [+01.9, -03.4], [-11.3, -00.2], [+03.9, +12.1],
932     [+08.8, -04.2], [+01.0, +18.7], [+06.2, +15.6], [-11.4, +00.3], [-07.0, +04.0],
933     ])
934 }
935
936 # long loop and a dense map
937 scenario_1 = {
938     'v': 1.5,
939     'w': 0.1,
940     'landmarks': np.array([
941         [+11.6, +26.9], [+12.9, +12.3], [-01.1, +31.4], [+12.8, +05.4], [-10.5, +25.2],
942         [-07.4, +27.3], [-13.1, +19.1], [+07.8, +02.7], [+11.1, +25.9], [-13.6, +11.2],
943         [-10.4, +25.6], [-15.6, +20.3], [+10.0, +25.6], [+09.8, +03.1], [+05.8, +02.8],
944         [-09.5, +27.7], [+01.9, +29.5], [+02.4, +00.8], [-12.7, +06.4], [+09.0, +25.5],
945         [+08.1, +04.3], [+01.4, +30.9], [+07.9, +25.4], [-12.6, +07.0], [-12.0, +09.0],
946         [+13.9, +20.6], [-13.8, +06.2], [+00.5, +31.6], [-08.6, +28.4], [+16.0, +13.5],
947     ])
948 }
949
950
951 # long loop and a sparse map
952 scenario_2 = {
953     'v': 1.5,
954     'w': 0.1,
955     'landmarks': np.array([
956         [+2.6, +2.7], [+4.4, -0.9], [-0.1, +1.3], [+6.9, -5.2], [-5.9, +5.7],
957         [-1.8, +3.0], [-2.2, +0.7], [+4.2, -6.6], [+4.3, +4.2], [-1.3, -0.4],
958     ])
959 }
960
961 scenario_3 = {
962     'v': 1.5,
963     'w': 0.1,
964     'landmarks': np.array([
965         [0.0, +7.5]
966     ])
967 }
968
969 # return
970 for scenario in [scenario_1]:
971     for R_constant in [1, 2, 4]:
972         execution(
973             landmarks=scenario['landmarks'],
974             v=scenario['v'],
975             w=scenario['w'],
976             landmarks_known=False,
977             R_constant=R_constant,
978             save=True,
979             show=False
980         )
981
982
983
984 if __name__ == '__main__':
985     main()

```