



IP PARIS

INSTITUT POLYTECHNIQUE DE PARIS
ENSTA PARIS

CSC_5RO12_TA, Navegation Robotique

TP3, Particle Filter

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
24 octobre 2024

Table des matières

1 Question 1	2
1.1 Structure du Code	2
1.2 Execution Initiale	3
1.2.1 Utilization	3
2 Question 2	4
2.1 PF	4
2.1.1 motion_model_prediction()	4
2.1.2 observation_model_prediction()	5
2.1.3 resample_particles()	5
2.1.4 low_variance_resample()	6
2.1.5 multinomial_resample()	6
2.1.6 P()	7
3 Question 3	8
4 Question 4	9
5 Question 5	10
6 Question 6	11
7 Question 7	12
8 Question 8	13
9 Question 9	14

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. PF : Particle Filter sous forme de dataclass, qui contient les méthodes suivantes :
 - (a) `motion_model_prediction()`;
 - (b) `observation_model_prediction()`;
 - (c) `resample_particles()`;
 - (d) `low_variance_resample()`;
 - (e) `multinomial_resample()`;
 - (f) `P()`;
2. Simulation : classe pour la création du environnement de simulation du robot ;
 - (a) `get_observation()` : retourne une mesure bruitée d'une amère aléatoire ;
 - (b) `get_odometry()` : retourne une mesure bruitée de l'odométrie et de la commande du robot ;
 - (c) `get_robot_control()` : retourne la véritable commande du robot ;
 - (d) `simulate_world()` : simule le système à l'instant k ;
3. Utils : classe pour stocker des fonctions d'assistance pour l'exécution de l'algorithme ;
 - (a) `compute_motion()` : calcule le mouvement du robot selon son équation de mouvement ;
 - (b) `convert_angle()` : rappelle un angle entre $[-\pi, +\pi]$;

Les méthodes de PF seront précisées dans la Question 2.

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.

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 :

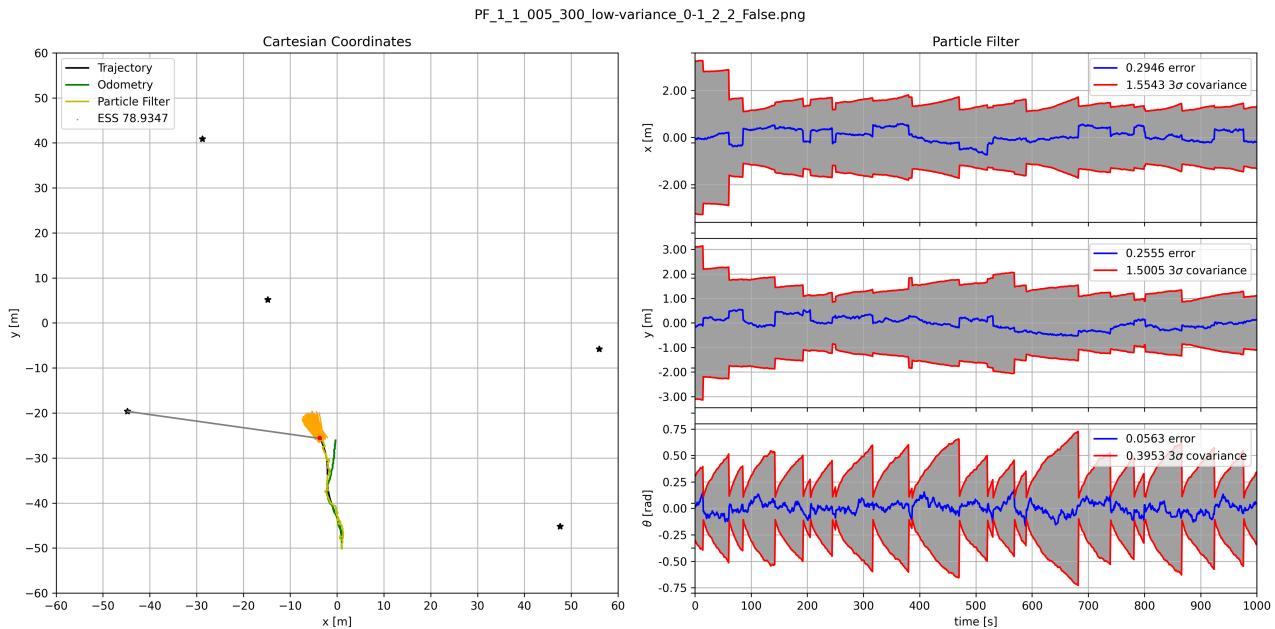


FIGURE 1.1 : Résultats Initiales

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

1. `j` : `dt_measurement` : intervalle entre deux mesures consécutives en secondes ;
2. `i` : `dt_prediction` : intervalle entre deux prédictions consécutives en secondes ;
3. `k` : `n_landmarks` : nombre de références sur la simulation ;
4. `l` : `n_particles` : nombre de particules sur le filtre ;
5. `m` : `resample_method` : méthode de rééchantillonnage ;
6. `n` : `Q_constant` : constant de la matrice de covariance du bruit du processus \mathbf{Q}_k ;
7. `o` : `R_constant` : constant de la matrice de covariance du bruit de mesure \mathbf{R}_k ;
8. `p` : `black_out` : absence de mesures entre 250 et 300 secondes ? ;

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     arr = [...]
10
11    for var in arr:
12        execution(var, show_result=True, save_result=False)
13
14 if __name__ == "__main__":
15    main()

```

2. Question 2

2.1. PF

Pour calculer le Filtre de Particules, une dataclass a été utilisée, car la classe ne contient que des méthodes, qui sont expliquées ci-dessous.

2.1.1. motion_model_prediction()

Définition 2.1. La prédiction du modèle de mouvement est donnée par l'équation suivante :

$$\hat{\mathbf{x}}_{k|k-1} = f(\hat{\mathbf{x}}_{k-1}, \tilde{\mathbf{u}}_k, \mathbf{w}_k) = \begin{bmatrix} x_{k-1} + ((\tilde{v}_k^x + w_k^{v_x}) \cos(\theta_{k-1}) - (\tilde{v}_k^y + w_k^{v_y}) \sin(\theta_{k-1}) \cdot \Delta t) \\ y_{k-1} + ((\tilde{v}_k^x + w_k^{v_x}) \sin(\theta_{k-1}) + (\tilde{v}_k^y + w_k^{v_y}) \cos(\theta_{k-1}) \cdot \Delta t) \\ \theta_{k-1} + (\tilde{\omega}_k + w_k^\omega) \Delta t \end{bmatrix} \quad (2.1)$$

Où :

1. **Robot State** : $\mathbf{x}_k = [x_k \ y_k \ \theta_k]^\top$ à l'instant k , relative à l'origine du plan cartésien.
2. **Noised Odometry** : $\tilde{\mathbf{u}}_k = [\tilde{v}_k^x \ \tilde{v}_k^y \ \tilde{\omega}_k]^\top \sim \mathcal{N}(\mathbf{u}_k, \mathbf{Q}_k)$ à l'instant k , relative au robot :
 - (a) **Robot Control** : $\mathbf{u}_k = [v_k^x \ v_k^y \ \omega_k]^\top$ à l'instant k , relative au robot.
 - (b) **Process Noise** : $\mathbf{w}_k = [w_k^{v_x} \ w_k^{v_y} \ w_k^\omega]^\top \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_k)$.
 - (c) **Process Noise Covariance** : \mathbf{Q}_k covariance du bruit Gaussian du processus.

La fonction `motion_model_prediction()` implémente la prédiction du modèle de mouvement :

```

1 def motion_model_prediction(
2     x: np.ndarray[float], u: np.ndarray[float], dt: float, Q_estimation: np.ndarray[float]
3 ) -> np.ndarray[float]:
4     """
5     Return motion model of agent.
6
7     Args:
8         x (np.ndarray[float]): system state at instant k-1.
9         u (np.ndarray[float]): control input, or odometry measurement, at instant k.
10        dt (float): simulation time step in seconds.
11        Q_estimation (np.ndarray[float]): process noise covariance matrix Q estimation.
12    """
13    x, y, theta = x[0, 0], x[1, 0], x[2, 0]
14    vx, vy, omega = u[0, 0], u[1, 0], u[2, 0]
15
16    w = np.random.multivariate_normal([0, 0, 0], Q_estimation)
17    w_vx, w_vy, w_omega = w[0], w[1], w[2]
18
19    x_prediction = np.array([
20        [x + ((vx + w_vx) * np.cos(theta) - (vy + w_vy) * np.sin(theta)) * dt],
21        [y + ((vx + w_vx) * np.sin(theta) + (vy + w_vy) * np.cos(theta)) * dt],
22        [theta + (omega + w_omega) * dt]
23    ])
24    x_prediction[2, 0] = Utils.convert_angle(x_prediction[2, 0])
25
26    return x_prediction

```

Listing 1 : `motion_model_prediction()`

Remarque. Ça definition correspond à la fonction `Utils.compute_motion()`.

2.1.2. `observation_model_prediction()`

Définition 2.2. La prédiction du modèle d'observation est donnée par l'équation suivante :

$$\hat{y}_k = h(\hat{x}_k) = \begin{bmatrix} \sqrt{(x_k^p - x_k)^2 + (y_k^p - y_k)^2} \\ \arctan\left(\frac{y_k^p - y_k}{x_k^p - x_k}\right) - \theta_k \end{bmatrix} \quad (2.2)$$

La fonction `observation_model_prediction()` implémente la prédiction du modèle d'observation :

```

1 def observation_model_prediction(
2     x: np.ndarray[float], i: int, landmarks: np.ndarray[float]
3 ) -> np.ndarray[float]:
4 """
5     Returns observation model prediction as np.ndarray[float] of system command y at instant k.
6
7     Args:
8         x (np.ndarray[float]): system state at instant k-1.
9         i (int): observed landmark index.
10        landmarks (np.ndarray[float]): coordinates x and y of all landmarks.
11    """
12    x, y, theta = x[0, 0], x[1, 0], x[2, 0]
13    x_i, y_i = landmarks[0, i], landmarks[1, i]
14
15    h = np.array([
16        [np.sqrt((x_i - x)**2 + (y_i - y)**2)],
17        [np.arctan2((y_i - y), (x_i - x)) - theta],
18    ])
19    h[1, 0] = Utils.convert_angle(h[1, 0])
20
21    return h

```

Listing 2 : `observation_model_prediction()`

2.1.3. `resample_particles()`

Définition 2.3. Le **rééchantillonnage à faible variance** sélectionne les particules les plus représentatives de l'état réel. Cela permet de mieux préserver les particules avec des poids élevés, réduisant ainsi la perte de diversité tout en minimisant la répétition excessive de certaines particules.

La fonction `resample_particles()` implémente le jacobian du modèle de prédiction de mouvement :

```

1 def resample_particles(
2     particles: np.ndarray[float], weights: np.ndarray[float], n: int, method: str =
3     'low-variance'
4 ) -> np.ndarray[float]:
5 """
6     Return resampled particles as np.ndarray[float] based on method
7
8     Args:
9         particles (np.ndarray[float]): particles states: x, y, theta.
10        weights_particles (np.ndarray[float]): particle weights.
11        n (int): number of particles.
12        method (str): resample method. Default is 'low-variance'.
13    """
14    match method.upper():
15        case 'LOW-VARIANCE':
16            return PF.low_variance_resample(particles, weights, n)
17
18        case 'MULTINOMIAL':
19            return PF.multinomial_resample(particles, weights, n)
20
21        case _:
22            return None

```

Listing 3 : `resample_particles()`

2.1.4. low_variance_resample()

Définition 2.4. La méthode de rééchantillonnage à faible variance consiste à sélectionner des particules de manière plus uniforme en fonction de leurs poids, en répartissant les rééchantillons avec un espace régulier, ce qui permet de minimiser le risque de dupliquer excessivement certaines particules tout en éliminant celles avec un faible poids.

```

1 def low_variance_resample(
2     particles: np.ndarray[float], weights: np.ndarray[float], n: int
3 ) -> np.ndarray[float]:
4 """
5     Return low-variance resampled particles based on their weights.
6
7     Args:
8         particles (np.ndarray[float]): particles states: x, y, theta.
9         weights_particles (np.ndarray[float]): particle weights.
10        n (int): number of particles.
11    """
12    base_indices = np.arange(0.0, 1.0, 1 / n)
13    random_offset = np.random.uniform(0, 1 / n)
14    resampling_indices = base_indices + random_offset
15
16    cumulative_weights = np.cumsum(weights)
17    cumulative_index = 0
18
19    resampled_indices = []
20    for i in range(n):
21        while resampling_indices[i] > cumulative_weights[cumulative_index]:
22            cumulative_index += 1
23
24        resampled_indices.append(cumulative_index)
25
26    resampled_particles = particles[:, resampled_indices]
27    resampled_weights = np.ones(n) / n
28
29    return resampled_particles, resampled_weights

```

Listing 4 : low_variance_resample()

2.1.5. multinomial_resample()

Définition 2.5. Le rééchantillonnage multinomial est une méthode utilisée dans les filtres particulaires pour sélectionner des particules en fonction de leurs poids. Chaque particule est tirée aléatoirement avec une probabilité proportionnelle à son poids, ce qui permet de privilégier les particules les plus représentatives de l'état actuel.

```

1 def multinomial_resample(
2     particles: np.ndarray[float], weights: np.ndarray[float], n: int
3 ) -> np.ndarray[float]:
4 """
5     Return multinomial resampled particles based on their weights.
6
7     Args:
8         particles (np.ndarray[float]): particles states: x, y, theta.
9         weights (np.ndarray[float]): particle weights.
10        n (int): number of particles.
11    """
12    indices = np.random.choice(np.arange(n), size=n, p=weights)
13
14    resampled_particles = particles[:, indices]
15    resampled_weights = np.ones(n) / n
16
17    return resampled_particles, resampled_weights

```

Listing 5 : multinomial_resample()

Remarque. Cependant, cette méthode peut entraîner une "impoverishment" des particules, où certaines sont sélectionnées plusieurs fois, réduisant ainsi la diversité de l'ensemble de particules.

2.1.6. P()

Définition 2.6. La matrice de covariance, dans le cadre d'un filtre Particulaire (PF), est noté \mathbf{P} et est estimée à partir de la distribution des particles et de leurs poids.

L'estimation de la matrice de covariance est donnée par l'équation suivante :

$$\hat{\mathbf{P}}_k = \sum_{i=1}^N w_k^i (\mathbf{x}_k^i - \hat{\mathbf{x}}_k) (\mathbf{x}_k^i - \hat{\mathbf{x}}_k)^T \quad (2.3)$$

La fonction P() implémente l'estimation de la mtrice de covariance :

```

1 def P(x_estimation: np.ndarray[float], particles: np.ndarray[float]) -> np.ndarray[float]:
2     """
3         Returns particle filter covariance matrix P estimation as np.ndarray[float].
4
5     Args:
6         x_estimation (np.array) : The mean/estimated state of the particles (3x1).
7         particles (np.array): An array of shape (3, N_particles) containing the state of all
8             particles (x, y, theta).
9     """
10    n = particles.shape[1]
11
12    P_estimation = np.zeros((3, 3))
13    for i in range(n):
14        deviation = (particles[:, i:i+1] - x_estimation).reshape(-1, 1)
15        deviation[2] = Utils.convert_angle(deviation[2])
16
17        P_estimation += deviation @ deviation.T
18    P_estimation /= n
19
20    return P_estimation

```

Listing 6 : P()

3. Question 3

Ci-dessous, quelques interactions ont été réalisées en faisant varier le bruit dynamique du filtre :

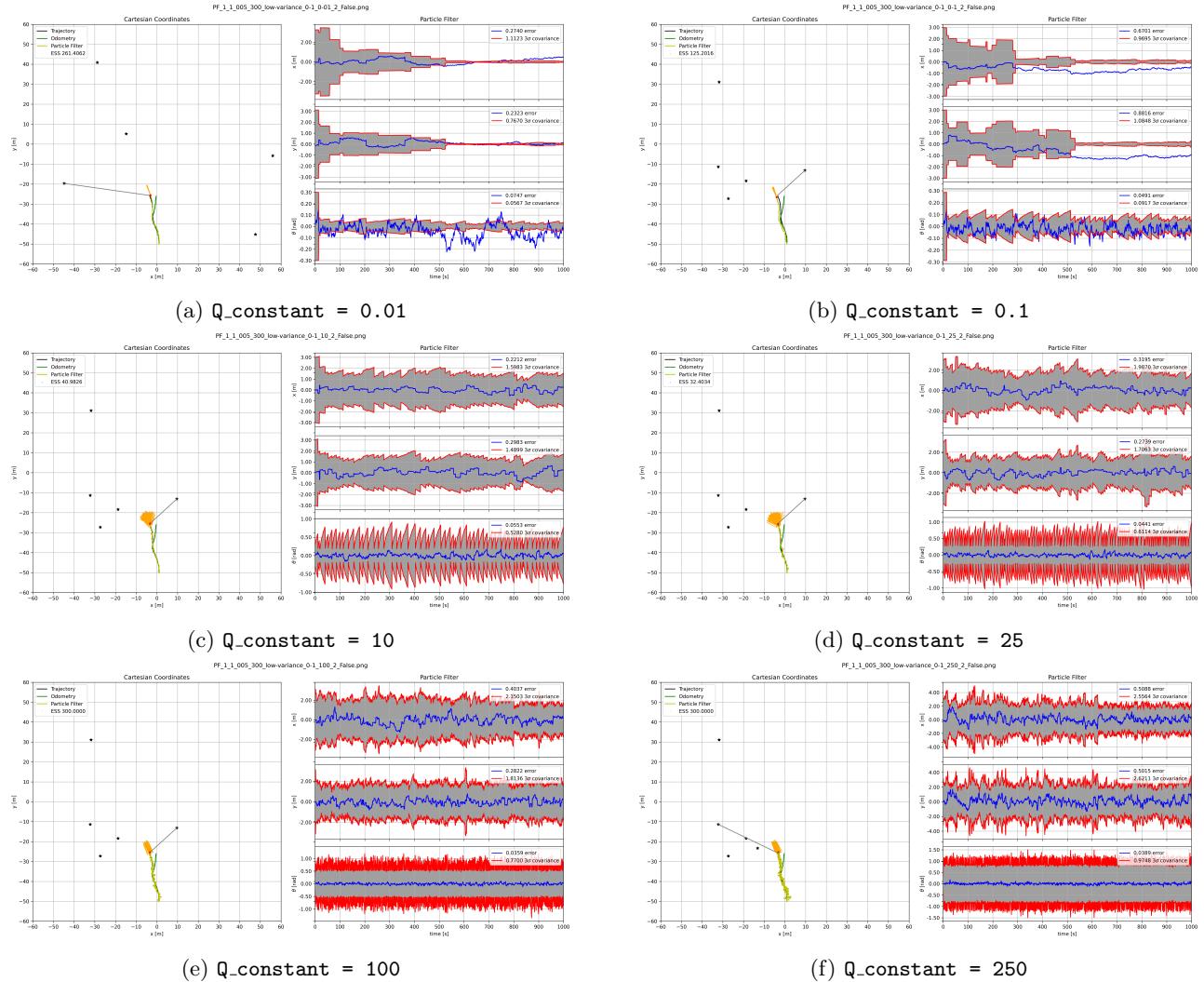


FIGURE 3.1 : Variance du bruit dynamique Q_{constant}

Il est à noter que lorsque la constante Q_{constant} est petite ($\approx Q_k < 1$) ou trop grande ($\approx Q_k > 5$), les résultats du filtre particulaire se dégradent. Dans le premier cas, le filtre souffre d'overfitting, car l'aire de la covariance ne prend pas en compte l'erreur. Dans le deuxième cas, un bruit excessif conduit à une estimation instable, même si celle-ci reste possible.

Cela montre à quel point l'efficacité du filtre particulaire, et la qualité de sa prédition de trajectoire, dépendent du bruit.

Remarque. Par ailleurs, il est notable que l'erreur et la covariance sont plus "instables" dans le filtre particulaire, en raison d'une plus grande variance entre les mesures consécutives. Ce comportement est attendu, car les particules ont des distributions indépendantes.

4. Question 4

Ci-dessous, quelques interactions ont été réalisées en faisant varier le bruit de mesure du filtre :

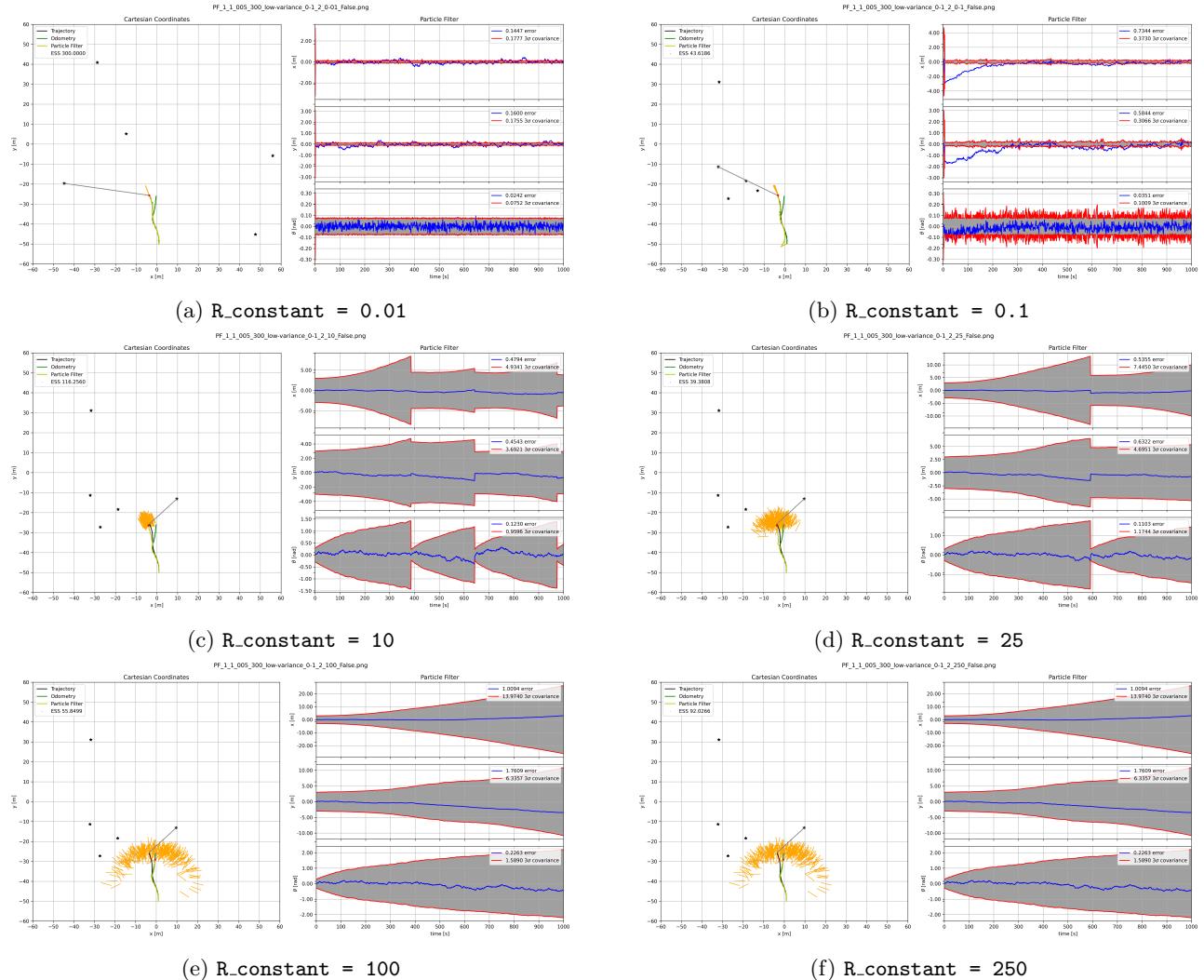


FIGURE 4.1 : Variance du bruit de mesure R_{constant}

Il est à noter que lorsque la constante R_{constant} est petite ($\approx R_k < 1$) ou trop grande ($\approx R_k > 5$), le résultat du filtre particulaire se dégradent. Dans le premier cas, il n'y a pas assez de variabilité entre les particules pour explorer différentes configurations, tandis que dans le deuxième cas, une variabilité excessive conduit à une estimation instable, car le filtre prend plus de temps pour procéder au rééchantillonnage et, par conséquent, pour corriger ses estimations.

Remarque. Lorsque le bruit augmente, le filtre perd son efficacité, car il ne corrige pas son estimation et suit simplement l'odométrie.

5. Question 5

Ci-dessous, quelques interactions ont été réalisées en faisant varier le seuil de ré-échantillonnage des poids en fonction de θ_{eff} du filtre :

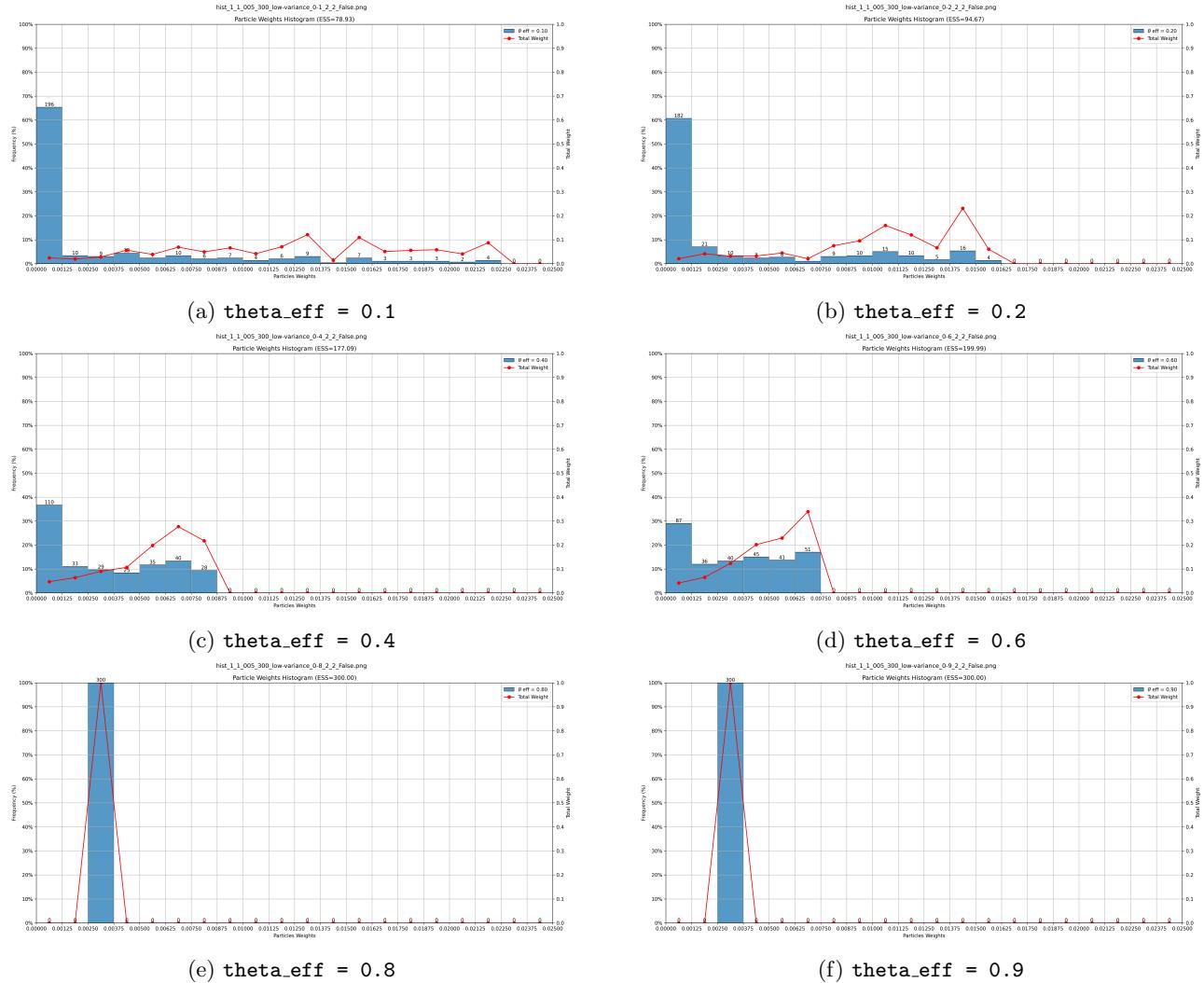


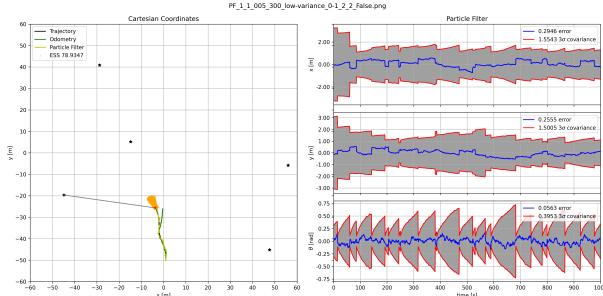
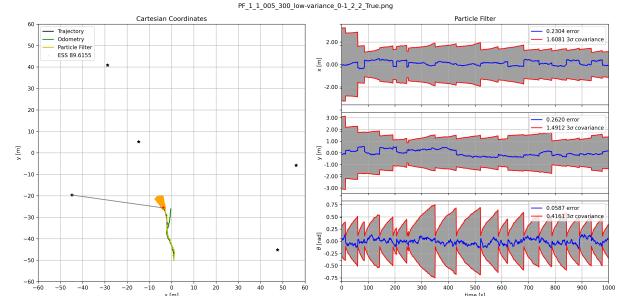
FIGURE 5.1 : Variance du seuil de ré-échantillonnage θ_{eff}

Il est à noter que lorsque θ_{eff} est trop petit ($\approx \theta_{\text{eff}} < 0.5$), la distribution totale des poids est assez uniforme, comme on peut l'observer avec la ligne rouge presque horizontale, car les rééchantillonnages sont plus probables. En revanche, lorsque θ_{eff} est trop grand ($\approx \theta_{\text{eff}} > 0.5$), la distribution totale des poids devient non uniforme, comme on peut voir avec la ligne rouge croissante, car les rééchantillonnages sont moins probables et entraînant ainsi un phénomène de dégénérescence.

Remarque. Lorsque θ_{eff} est trop grand ($\approx \theta_{\text{eff}} > 0.5$) le rééchantillonnage n'a pas lieu, ce qui laisse la dégénérescence intacte. Les occurrences se maintiennent entre 0.00250 et 0.00375, car le poid de chaque particle est initialisé à $1/300 \approx 0.0033$.

6. Question 6

Ci-dessous, un trou de mesures simulé entre 250 et 350 secondes :

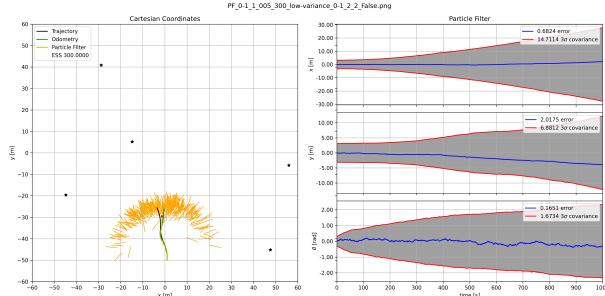
(a) `black_out = False`(b) `black_out = True`FIGURE 6.1 : Simulation d'un trou `black_out`

Il est à noter que lorsque il y a un trou de mesures entre 250 et 350 secondes peu de changements sur l'estimation peu être apperçu. Si, il y a une augmentation de l'erreur et de la covariance, plus notable à la covariance de θ , mais ce manque de données n'a pas beaucoup impacté le résultat du filtre particulaire.

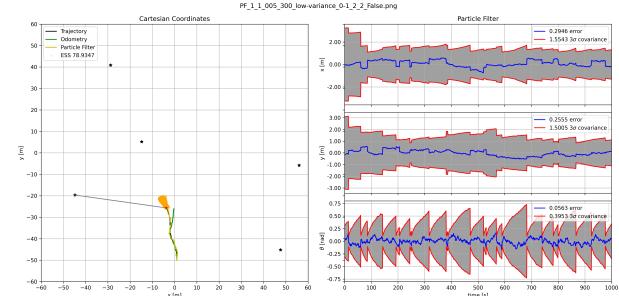
Remarque. Le filtre particulaire reste robuste face à une perte de mesures car il continue d'estimer l'état grâce au modèle prédictif et à la diversité des particules. Lorsque les mesures reviennent, le rééchantillonnage ajuste rapidement les particules, permettant une récupération efficace.

7. Question 7

Ci-dessous, quelques interactions ont été réalisées en faisant varier le nombre d'amarres sur la carte :



(a) `dt_measurement = 0.1`



(b) `dt_measurement = 1`

FIGURE 7.1 : Variance du bruit de mesure `dt_measurement`

Il est à noter que lorsque la fréquence de mesure augmente, c'est-à-dire lorsque `dt_measurement` se réduit, le filtre particulaire diverge, car les covariances augmentent de manière constante pendant l'exécution de l'algorithme, tandis que les particules s'éloignent. Ce comportement est attendu, car la fréquence de mesure est supérieure à la fréquence de prédiction, ce qui empêche le filtre d'être appliqué efficacement.

Remarque. Lorsque le bruit augmente, le filtre perd son efficacité, car il ne corrige pas son estimation et suit simplement l'odométrie.

8. Question 8

Ci-dessous, quelques interactions ont été réalisées en faisant varier le nombre d'amers sur la carte :

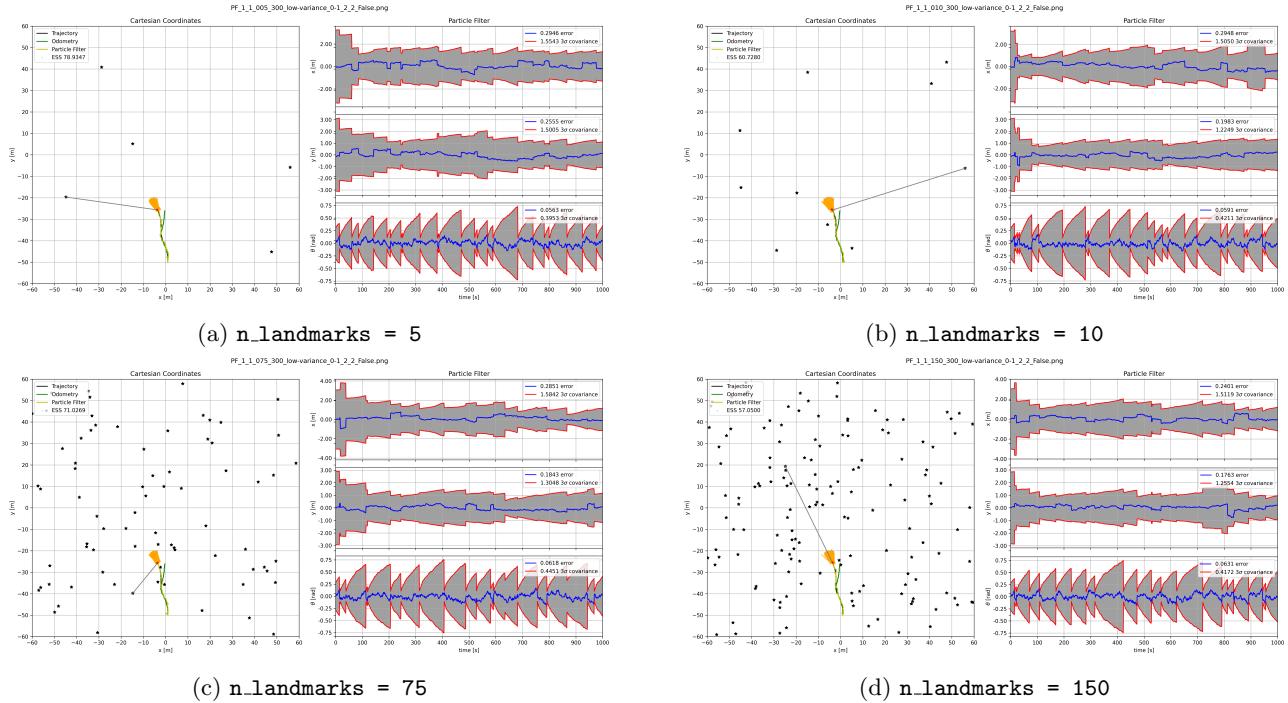


FIGURE 8.1 : n_landmarks

Il est à noter que lorsque la quantité de références augmente il n'a pas beaucoup d'influence sur le résultat du filtre particulaire car l'erreur, la covariance et la trajectoire ne présentent pas des changements significatifs.

Tant que les particules sont bien réparties et que le modèle de capteur relie avec précision les mesures aux amers, le filtre peut fonctionner efficacement avec un nombre réduit d'amers en se concentrant sur les mises à jour probabilistes, plutôt que sur un grand ensemble de points de référence.

Remarque. Si le nombre d'amers est limité, le filtre ajuste les poids des particules en conséquence, conservant ainsi une performance efficace sans nécessiter un grand nombre de points de référence. Toutefois, un nombre accru d'amers peut améliorer la précision en réduisant l'incertitude de l'estimation, mais l'algorithme reste robuste avec peu d'amers grâce à sa nature probabiliste.

9. Question 9

Ci-dessous, la méthode de rééchantillonnage multinomial a été choisie pour la fonction `resample_method` :

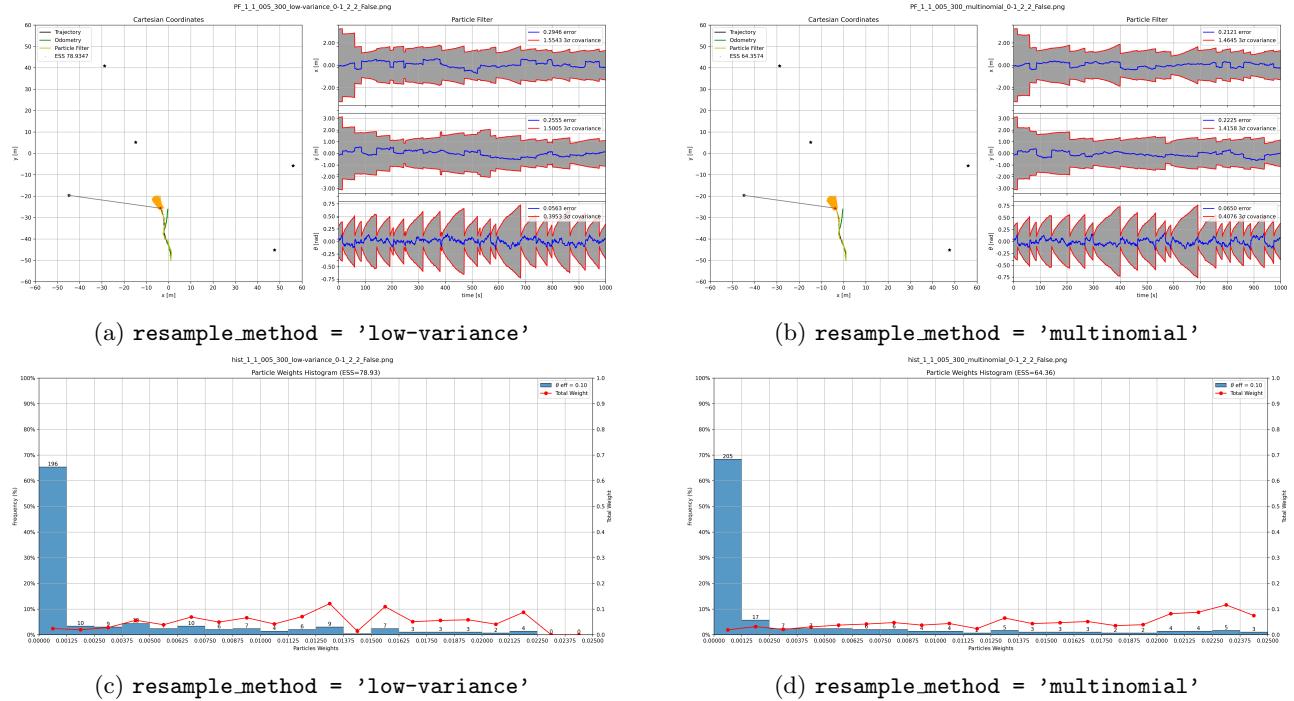


FIGURE 9.1 : Comparaison des `resample_method`

La méthode originale privilégie les particules à faible variance, ce qui réduit le facteur aléatoire dans la sélection des particules et rend le filtre plus efficace en permettant une convergence plus rapide. Pour une approche plus simple et plus aléatoire, la méthode multinomiale a été choisie, car son implémentation est plus facile à mettre en oeuvre et les particules sont sélectionnées de manière totalement aléatoire.

Ce changement de méthode améliore légèrement l'efficacité du filtre particulaire, car la dégénérescence est réduite, comme on peut le constater dans l'histogramme plus régulier pour la méthode multinomiale. De plus, l'erreur et la covariance diminuent pour les coordonnées x et y avec cette nouvelle méthode, bien que l'angle θ ne s'améliore pas de manière significative.

Remarque. D'autres méthodes peuvent être choisies en fonction de l'application. Dans le cas d'une trajectoire relativement stable, comme celle étudiée ici, les deux méthodes offrent des résultats satisfaisants.