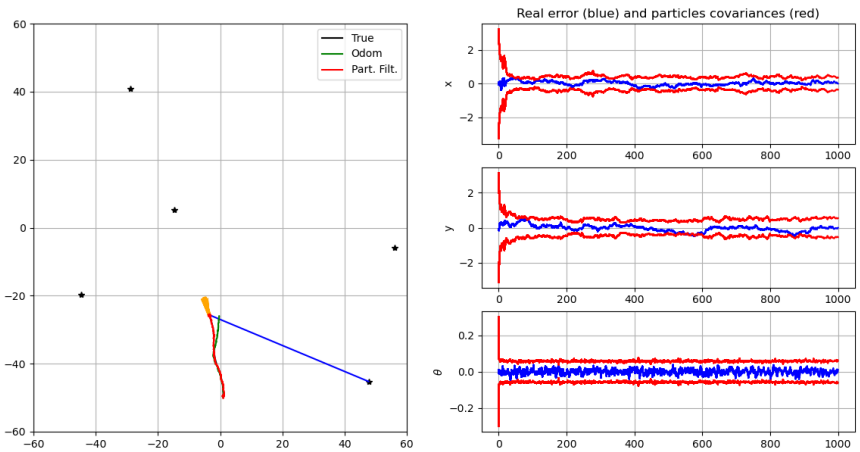
TP3 : Filtrage particulaire

ROB312 - Navigation pour les systèmes autonomes

Bastien HUBERT



ENSTA Paris - octobre 2022

1 Q1

Le code ci-dessous correspond au code Python d'un filtrage particulaire appliqué au déplacement d'un robot. Il se décompose en différentes parties :

1.1 Simulation

Cette partie correspond à toutes les fonction nécessaires à la simulation du robot (génération de la trajectoire, acquisition de l'odométrie, acquisition des données capteurs).

```
"""
TP particle filter for mobile robots localization

authors: Goran Frehse, David Filliat, Nicolas Merlinge
"""

from math import sin, cos, atan2, pi
import matplotlib.pyplot as plt
import numpy as np
seed = 123456
np.random.seed(seed)

from scipy.linalg import sqrtm

import os
try:
    os.makedirs("outputs")
except:
    pass


# ----- Simulator class (world, control and sensors) -----

class Simulation:
    def __init__(self, Tf, dt_pred, xTrue, QTrue, xOdom, Map, RTrue, dt_meas):
        self.Tf = Tf
        self.dt_pred = dt_pred
        self.nSteps = int(np.round(Tf/dt_pred))
        self.QTrue = QTrue
        self.xTrue = xTrue
        self.xOdom = xOdom
        self.Map = Map
        self.RTrue = RTrue
```

```
        self.dt_meas = dt_meas

    # return true control at step k
    def get_robot_control(self, k):
        # generate  sin trajectory
        u = np.array([[0, 0.025,  0.1*np.pi / 180 * sin(3*np.pi * k / self.nSteps)]]).T
        return u


    # simulate new true robot position
    def simulate_world(self, k):
        dt_pred = self.dt_pred
        u = self.get_robot_control(k)
        self.xTrue = tcomp(self.xTrue, u, dt_pred)
        self.xTrue[2, 0] = angle_wrap(self.xTrue[2, 0])


    # computes and returns noisy odometry
    def get_odometry(self, k):
        # Ensuring random repetability for given k
        np.random.seed(seed*2 + k)

        # Model
        dt_pred = self.dt_pred
        u = self.get_robot_control(k)
        xnow = tcomp(self.xOdom, u, dt_pred)
        uNoise = np.sqrt(self.QTrue) @ np.random.randn(3)
        uNoise = np.array([uNoise]).T
        xnow = tcomp(xnow, uNoise, dt_pred)
        self.xOdom = xnow
        u = u + dt_pred*uNoise
        return xnow, u


    # generate a noisy observation of a random feature
    def get_observation(self, k):
        # Ensuring random repetability for given k
        np.random.seed(seed*3 + k)

        # Model
        if k*self.dt_pred % self.dt_meas == 0:
            notValidCondition = False # False: measurement valid / True: measurement not vali
            if notValidCondition:
```

```
                z = None
                iFeature = None
            else:
                iFeature = np.random.randint(0, self.Map.shape[1] - 1)
                zNoise = np.sqrt(self.RTrue) @ np.random.randn(2)
                zNoise = np.array([zNoise]).T
                z = observation_model(self.xTrue, iFeature, self.Map) + zNoise
                z[1, 0] = angle_wrap(z[1, 0])
        else:
            z = None
            iFeature = None
        return [z, iFeature]
```

1.2 Fonctions de prédiction et d'observation

Cette partie modélise la dynamique de déplacement du robot, ainsi que l'observation d'un amère depuis le robot.

```
# ---- Particle Filter: model functions ----


# evolution model (f)
def motion_model(x, u, dt_pred):
    # x: estimated state (x, y, heading)
    # u: control input (Vx, Vy, angular rate)

    [x_avant, y_avant, theta_avant] = x[:, 0]
    [vx, vy, omega] = u[:, 0]

    x_apres = x_avant + (vx * cos(theta_avant) - vy * sin(theta_avant)) * dt_pred
    y_apres = y_avant + (vx * sin(theta_avant) + vy * cos(theta_avant)) * dt_pred
    theta_apres = theta_avant + omega * dt_pred

    xPred = [[x_apres],
             [y_apres],
             [theta_apres]]

    return np.array(xPred)


# observation model (h)
def observation_model(xVeh, iFeature, Map):
    # xVeh: vecule state
    # iFeature: observed amer index
```

```
    # Map: map of all amers
    [x, y, theta] = xVeh[:, 0]
    [xP, yP] = Map[:, iFeature]

    z = [[np.sqrt( (xP - x)**2 + (yP - y)**2 )],
         [atan2(yP - y, xP - x) - theta]]

    return np.array(z)
```

1.3 Ré-échantillonnage

Cette partie correspond à la fonction de ré-échantillonnage qui a lieu lorsque trop de particules ont un poids faible, et ne sont pas donc représentatives d'états probables du système (ie de position vraisemblable du robot).

```
# ---- particle filter implementation ----

# Particle filter resampling
def re_sampling(px, pw):
    """
    low variance re-sampling
    """

    w_cum = np.cumsum(pw)
    base = np.arange(0.0, 1.0, 1 / nParticles)
    re_sample_id = base + np.random.uniform(0, 1 / nParticles)
    indexes = []
    ind = 0
    for ip in range(nParticles):
        while re_sample_id[ip] > w_cum[ind]:
            ind += 1
        indexes.append(ind)

    px = px[:, indexes]
#     pw = pw[indexes]

    # Normalization
    pw = np.ones(pw.shape)
    pw = pw / np.sum(pw)

    return px, pw
```

1.4 Fonctions auxiliaires

Cette partie correspond à des fonctions auxiliaires pour le calcul et l'affichage du filtre. Notons l'ajout de la fonction *reformat*, qui a pour but de convertir xParticles en un tableau numpy de matrices colonne, dont les éléments sont directement lisibles par les fonctions de la section 1.2.

```
# ---- Utils functions ----

# return the coordinates of particle p in x in the format of a column matrix
def reformat( x, p ):
    return np.array( [[x[0, p]],
                      [x[1, p]],
                      [x[2, p]]] )


# Init displays
show_animation = True
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True, figsize=(14, 7))
ax3 = plt.subplot(3, 2, 2)
ax4 = plt.subplot(3, 2, 4)
ax5 = plt.subplot(3, 2, 6)


# fit angle between -pi and pi
def angle_wrap(a):
    if (a > np.pi):
        a = a - 2 * pi
    elif (a < -np.pi):
        a = a + 2 * pi
    return a


# composes two transformations
def tcomp(tab, tbc, dt):
    assert tab.ndim == 2 # eg: robot state [x, y, heading]
    assert tbc.ndim == 2 # eg: robot control [Vx, Vy, angle rate]
    #dt : time-step (s)

    angle = tab[2, 0] + dt * tbc[2, 0] # angular integration by Euler

    angle = angle_wrap(angle)
    s = sin(tab[2, 0])
    c = cos(tab[2, 0])
    position = tab[0:2] + dt * np.array([[c, -s], [s, c]]) @ tbc[0:2] # position integration
```

```
    out = np.vstack((position, angle))

    return out


def plotParticles(simulation, k, iFeature, hxTrue, hxOdom, hxEst, hxError, hxSTD, save = True
    # simulation : Simulation object (containing world simulation and sensors)
    # k : current time-step
    # iFeature : index of current emitting amer
    # hxTrue : true trajectory
    # hxOdom : odometric trajectory
    # hxEst : estimated trajectory
    # hxError : error (basically "hxEst - hxTrue")
    # hxSTD : standard deviation on estimate
    # save : True to save a figure as an image

    # 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])

    ax1.cla()

    # Plot true landmark and trajectory
    ax1.plot(simulation.Map[0, :], simulation.Map[1, :], "*k")
    ax1.plot(hxTrue[0, :], hxTrue[1, :], "-k", label="True")
    if iFeature != None: ax1.plot([simulation.xTrue[0][0], simulation.Map[0, iFeature]], [sim

    # Plot odometry trajectory
    ax1.plot(hxOdom[0, :], hxOdom[1, :], "-g", label="Odom")

    # Plot estimated trajectory and current particles
    ax1.plot(hxEst[0, :], hxEst[1, :], "-r", label="Part. Filt.")
    ax1.plot(xEst[0], xEst[1], ".r")
    ax1.scatter(xParticles[0, :], xParticles[1, :], s=wp*10)
    for i in range(nParticles):
        ax1.arrow(xParticles[0, i], xParticles[1, i], 5*np.cos(xParticles[2, i]+np.pi/2), 5*n

    ax1.axis([-60, 60, -60, 60])
    ax1.grid(True)
    ax1.legend()

    # plot errors curves
    ax3.plot(hxError[0, :], 'b')
```

```
    ax3.plot( 3.0 * hxSTD[0, :], 'r')
    ax3.plot(- 3.0 * hxSTD[0, :], 'r')
    ax3.grid(True)
    ax3.set_ylabel('x')
    ax3.set_title('Real error (blue) and particles covariances (red)')


    ax4.plot(hxError[1, :], 'b')
    ax4.plot( 3.0 * hxSTD[1, :], 'r')
    ax4.plot(- 3.0 * hxSTD[1, :], 'r')
    ax4.grid(True)
    ax4.set_ylabel('y')

    ax5.plot(hxError[2, :], 'b')
    ax5.plot( 3.0 * hxSTD[2, :], 'r')
    ax5.plot(- 3.0 * hxSTD[2, :], 'r')
    ax5.grid(True)
    ax5.set_ylabel(r"$\theta$")

    plt.figure(2)
    plt.hist(wp)
    plt.xlabel('Number of Particles')
    plt.ylabel('Weight')
    plt.title('Weight distribution', fontsize=10)

    if save: plt.savefig(r'outputs/SRL' + str(k) + '.png')
#         plt.pause(0.01)
```

1.5 Programme principal

Cette partie correspond au filtre particulaire à proprement parler, avec initialisation du filtre, itérations des observations, prédictions et corrections, et affichage des trajectoires réelle, estimées par l'odométrie et corrigée par le filtre.

```
# ============================================================
# Main Program
# ============================================================

# Enable/disable plotting
is_plot = True

# Nb of particle in the filter
nParticles = 300
```

```
# Simulation time
Tf = 1001          # final time (s)
dt_pred = 1        # Time between two dynamical predictions (s)
dt_meas = 1        # Time between two measurement updates (s)

# Location of landmarks
nLandmarks = 5
Map = 120*np.random.rand(2, nLandmarks)-60

# True covariance of errors used for simulating robot movements
QTrue = np.diag([0.02, 0.02, 1*pi/180]) ** 2
RTrue = np.diag([0.5, 1*pi/180]) ** 2

# Modeled errors used in the Particle filter process
QEst = 2 * np.eye(3, 3) @ QTrue
REst = 2 * np.eye(2, 2) @ RTrue

# initial conditions
xTrue = np.array([[1, -50, 0]]).T
#xTrue = np.array([[1, -40, -pi/2]]).T
xOdom = xTrue

# initial conditions: - a point cloud around truth
xParticles = xTrue + np.diag([1, 1, 0.1]) @ np.random.randn(3, nParticles)

# initial conditions: global localization
#xParticles = 120 * np.random.rand(3, nParticles)-60

# initial weights
wp = np.ones((nParticles))/nParticles
wp = wp / np.sum(wp)

# initial estimate
xEst = np.average(xParticles, axis=1, weights=wp)
xEst = np.expand_dims(xEst, axis=1)
xSTD = np.sqrt(np.average((xParticles-xEst)*(xParticles-xEst),
                axis=1, weights=wp))
xSTD = np.expand_dims(xSTD, axis=1)

# Init history matrixes
hxEst = xEst
hxTrue = xTrue
```

```
hxOdom = xOdom
err = xEst - xTrue
err[2, 0] = angle_wrap(err[2, 0])
hxError = err
hxSTD = xSTD

# Simulation environment
simulation = Simulation(Tf, dt_pred, xTrue, QTrue, xOdom, Map, RTrue, dt_meas)

if is_plot: plotParticles(simulation, 0, None, hxTrue, hxOdom, hxEst, hxError, hxSTD, save =

# Temporal loop
REstInv = np.linalg.inv( REst )
for k in range(1, simulation.nSteps):
    print(k)
    # Simulate robot motion
    simulation.simulate_world(k)

    # Get odometry measurements
    xOdom, u_tilde = simulation.get_odometry(k)

    # do prediction
    # for each particle we add control vector AND noise:
    noise = np.random.multivariate_normal(np.zeros(3), QEst, nParticles)
    for p in range(nParticles):
        xParticles[:, p] = motion_model(xParticles[:, p:p+1],
                           u_tilde + noise[p], dt_pred)[:, 0]

    # reformat particles
    x = [reformat(xParticles, p) for p in range(nParticles)]

    # observe a random feature
    [z, iFeature] = simulation.get_observation(k)
    if z is not None:
        for p in range(nParticles):
            # Predict observation from the particle position
            zPred = observation_model(x[p], iFeature, Map)

            # Innovation : perception error
            Innov = z - zPred
            Innov[1] = angle_wrap(Innov[1])

            # Compute particle weight using gaussian model
```

```
            wp[p] = wp[p] * np.exp( -1/2 * (Innov.T @ REstInv @ Innov)[0][0] )
    # Normalization
    wp = wp / np.sum( wp )


    # Compute position as weighted mean of particles
    xEst = np.average(xParticles, axis=1, weights=wp)
    xEst = np.expand_dims(xEst, axis=1)

    # Compute particles std deviation
    xSTD = np.sqrt(np.average((xParticles-xEst)*(xParticles-xEst),
                    axis=1, weights=wp))
    xSTD = np.expand_dims(xSTD, axis=1)

    # Resampling
    theta_eff = 0.5
    Nth = nParticles * theta_eff

    Neff = 1 / np.sum( [wp[p]**2 for p in range( nParticles )] )

    if Neff < Nth:
        # Particle resampling
        xParticles, wp = re_sampling( xParticles, wp )

    # store data history
    hxTrue = np.hstack((hxTrue, simulation.xTrue))
    hxOdom = np.hstack((hxOdom, simulation.xOdom))
    hxEst = np.hstack((hxEst, xEst))
    err = xEst - simulation.xTrue
    err[2, 0] = angle_wrap(err[2, 0])
    hxError = np.hstack((hxError, err))
    hxSTD = np.hstack((hxSTD, xSTD))

    # plot every 20 updates
    if is_plot and k*simulation.dt_pred % 20 == 0:
        plotParticles(simulation, k, iFeature, hxTrue, hxOdom, hxEst,
                      hxError, hxSTD, save = True)

tErrors = np.sqrt(np.square(hxError[0, :]) + np.square(hxError[1, :]))
print("Mean (var) translation error: {:e} ({:e})".format(np.mean(tErrors), np.var(tErrors)))
print("Press Q in figure to finish...")
plt.show()
```

2 Q2

Le code complété est présenté aux sections 1.2 à 1.5. L'exécution de ce code conduit à la figure 1 :

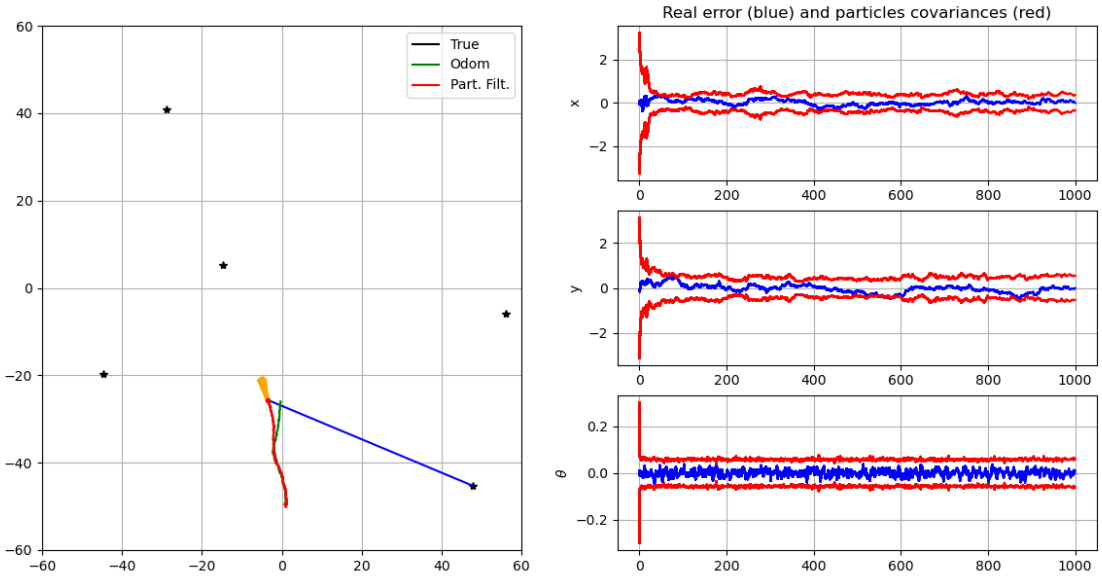


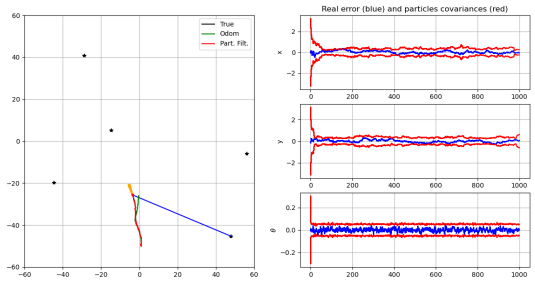
Figure 1: Filtrage particulaire

On remarque que la trajectoire corrigée par le filtre est très proche de la trajectoire réelle malgré la dérive importante de l'odométrie du robot. De plus, l'incertitude sur la position réduit très vite lors des premières itérations du filtre, puis se stabilise en régime permanent.

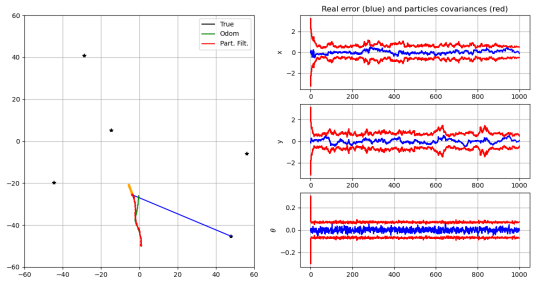
En comparant avec le filtre de Kalman, on remarque un temps d'exécution bien plus long car, si on ne calcule plus de jacobien et qu'on n'opère qu'une seule inversion matricielle (REst ne change pas au cours du temps ni entre les particules donc on peut calculer son inverse une seule fois et stocker la matrice inverse en mémoire), on effectue en revanche bien plus de multiplications matricielles et de produits scalaires. Les résultats obtenus sont toutefois similaires, car bien qu'il est difficile de comparer les nuages de points du filtre particulaire avec les ellipses de covariance du filtre de Kalman, les courbes d'erreurs et de covariances présentant les mêmes tendances et amplitudes pour les deux algorithmes.

3 Q3

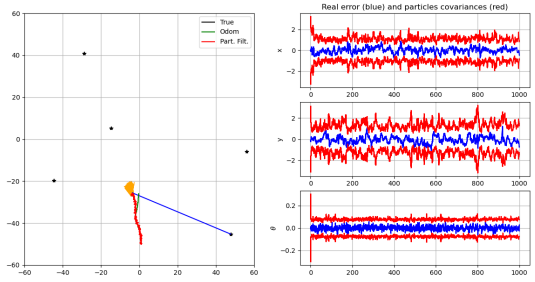
Les figures 2a à 2d correspondent à différentes trajectoires calculées par le filtre particulaire en fonction de la valeur de $QEst$, qui traduit le bruit dynamique du filtre, c'est-à-dire la confiance que l'on a dans l'estimation de la position du robot par son odométrie. Un coefficient multiplicatif important devant $QEst$ signifie que l'odométrie n'est pas fiable (par rapport aux mesures), ce qui se traduit par une trajectoire plus saccadée, et une covariance plus importante et variant plus. On constate par ailleurs que le nuage de particules grandit avec $QEst$, ce qui est simplement dû au fait que le bruit ajouté à l'étape de prédiction est gaussien d'écart-type $QEst$.



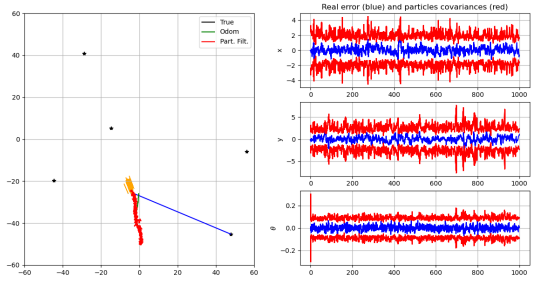
(a) $QEst = 1 * np.eye(3,3)$ @ $QTrue$



(b) $QEst = 10 * np.eye(3,3)$ @ $QTrue$



(c) $QEst = 100 * np.eye(3,3)$ @ $QTrue$

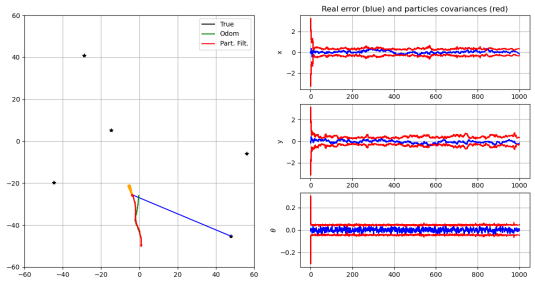


(d) $QEst = 1000 * np.eye(3,3)$ @ $QTrue$

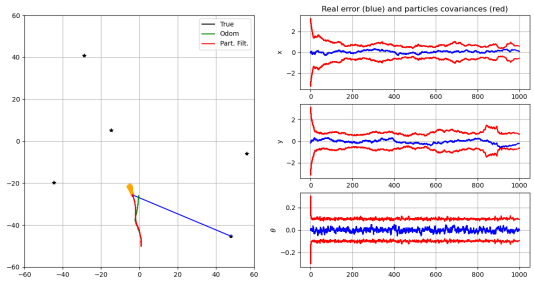
Figure 2: Filtrage particulaire pour différentes valeurs de $QEst$

4 Q4

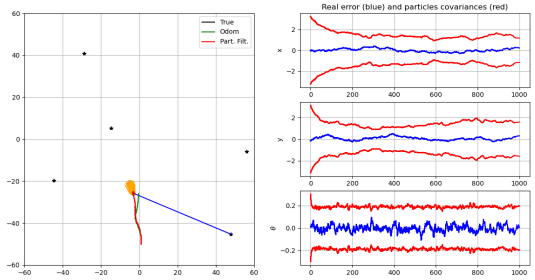
Les figures 3a à 3d correspondent à différentes trajectoires calculée par le filtre particulaire en fonction de la valeur de $REst$, qui traduit le bruit de mesure du filtre, c'est-à-dire la confiance que l'on a dans la mesure de la position du robot par rapport à un amère. Un coefficient multiplicatif important devant $REst$ signifie que les mesures ne sont pas fiables (par rapport à l'odométrie), ce qui se traduit par une trajectoire plus lisse car moins "micro-corrigée" par les amères, une covariance plus importante et réduisant moins souvent, et un nuage de particules moins dense.



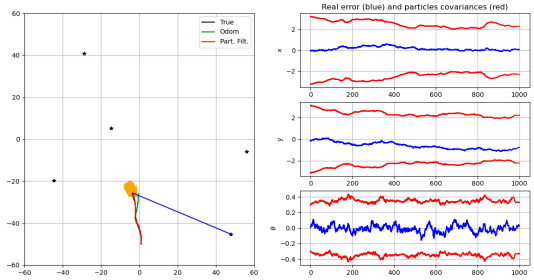
(a) $REst = 1 * np.eye(2,2)$ @ $RTrue$



(b) $REst = 10 * np.eye(2,2)$ @ $RTrue$



(c) $REst = 100 * np.eye(2,2)$ @ $RTrue$



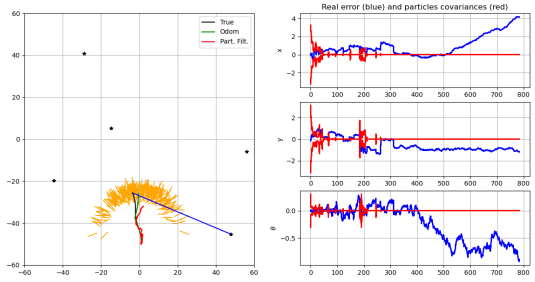
(d) $REst = 1000 * np.eye(2,2)$ @ $RTrue$

Figure 3: Filtre particulaire pour différentes valeurs de $REst$

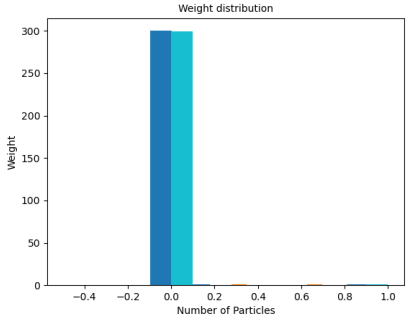
Mathématiquement, une matrice $REst$ dont les coefficients sont importants ramène l'exposant des coefficients de mise à jour des poids $-\frac{1}{2} Innov^T \; REst^{-1} \; Innov$ vers 0, pour obtenir dans le cas extrême $\forall i$, $\tilde{w}_k^i \approx w_{k-1}^i$. Dans ce cas, les particules sont uniformément pondérées, et la perception des amères n'est pas assez fiable pour réduire suffisamment le poids des particules incohérentes et correctement estimer la position et l'orientation du robot.

5 Q5

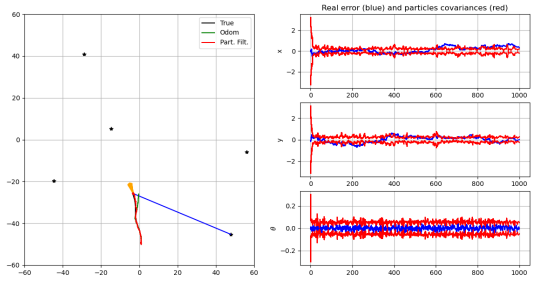
La dégénérescence du filtre particulaire est le cas extrême où toutes les particules ont un poids nul, sauf une qui a un poids de 1. L'estimation de l'état du système se fait alors uniquement par cette seule particule, tandis que les autres n'apportent aucune pondération et ne font que nécessiter du temps de calcul. Pour éviter ce problème, on fait appel à un ré-échantillonnage des particules quand trop peu d'entre elles jouent un rôle dans l'estimation de l'état du système, ce qui consiste à supprimer les particules correspondant à des états peu probables et à dupliquer les autres. Les figures 4a à 5h correspondent au filtrage particulaire pour différentes valeurs du seuil θ_{eff}.



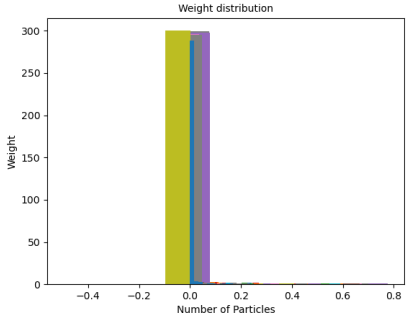
(a) $\theta_{eff} = 0,001$



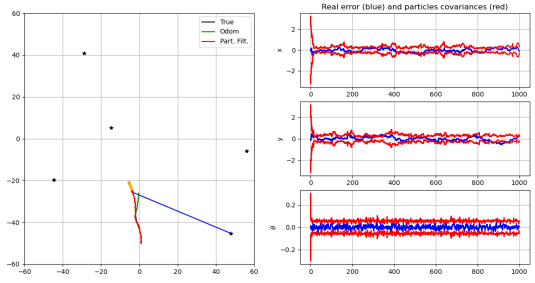
(b) Histogramme des poids correspondant



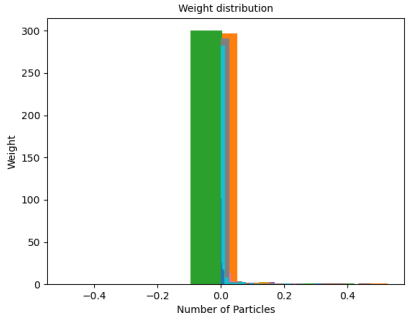
(c) $\theta_{eff} = 0,005$



(d) Histogramme des poids correspondant



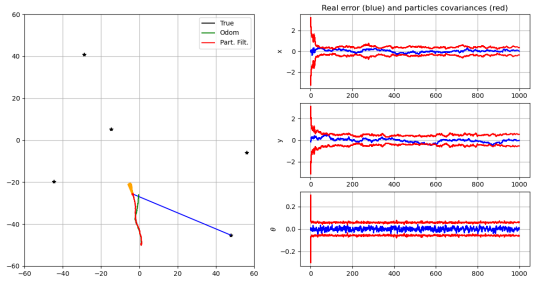
(e) $\theta_{eff} = 0,01$



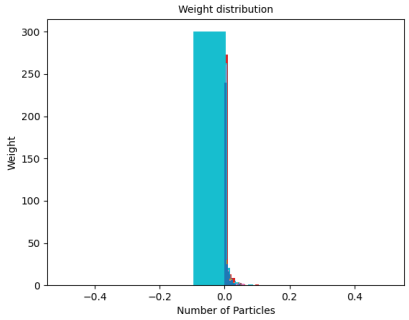
(f) Histogramme des poids correspondant

Figure 4: Filtre particulaire pour différentes valeurs de θ_{eff}

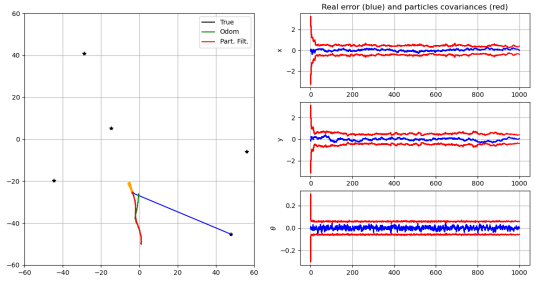
Pour θ_{eff} très petit, il n'y a pas de ré-échantillonnage, et on voit que les poids sont très inégalement répartis, avec principalement tous proches de 0 à de très rares exceptions. Ceci correspond au cas dégénéré que nous souhaitons éviter avec le ré-échantillonnage.



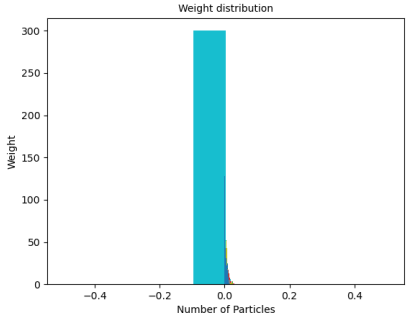
(a) $\theta_{eff} = 0,1$



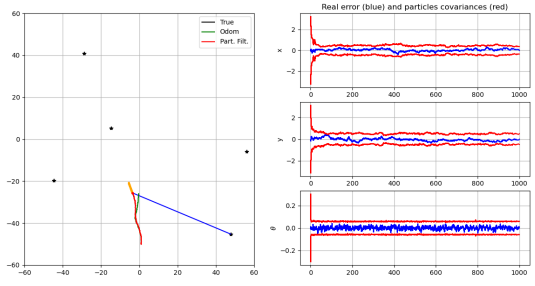
(b) Histogramme des poids correspondant



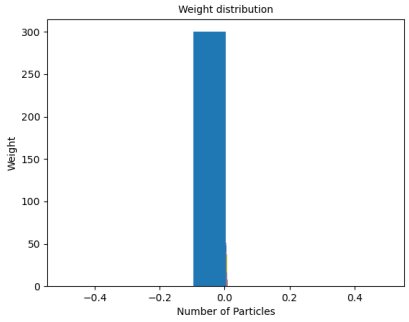
(c) $\theta_{eff} = 1/3$



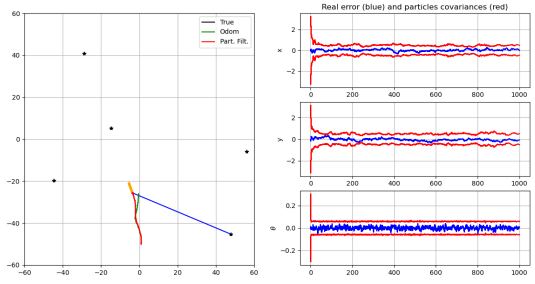
(d) Histogramme des poids correspondant



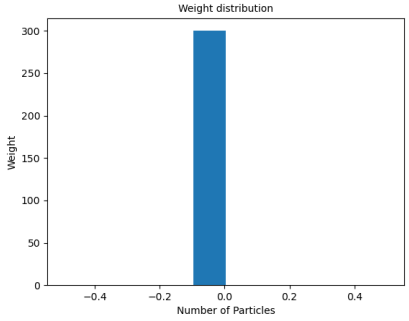
(e) $\theta_{eff} = 2/3$



(f) Histogramme des poids correspondant



(g) $\theta_{eff} = 1$



(h) Histogramme des poids correspondant

Figure 5: Filtre particulaire pour différentes valeurs de θ_{eff}

À l'inverse, pour θ_{eff} trop grand, le constant ré-échantillonnage fait que les poids sont tous uniformisés à 1/nParticles, ce qui est contre-productif: il n'y a que pour des valeurs proches de 1/3 que le ré-échantillonnage donne de bons résultats et que la répartition des poids corresponde à un nombre suffisant de particules "intéressantes".

6 Q6

On simule un trou de mesure en rajoutant la condition $notValidCondition = True$ pour $k \in [\![250, 350]\!]$ dans le code de *get_observation* de la section 1.1. L'exécution du programme avec cette perte des mesures est illustrée par la figure 6. L'odométrie étant l'unique donnée à disposition du filtre, celui-ci est incapable de corriger la trajectoire du robot : la trajectoire donnée est donc uniquement basée sur l'odométrie, qui dérive et dont l'incertitude augmente avec les accumulations d'erreurs. Quand les mesures sont de nouveau disponibles, le filtre corrige presque instantanément ces erreurs, et on retombe sur la trajectoire réelle sans que la période de trou n'ai d'impact à long terme sur le filtre.

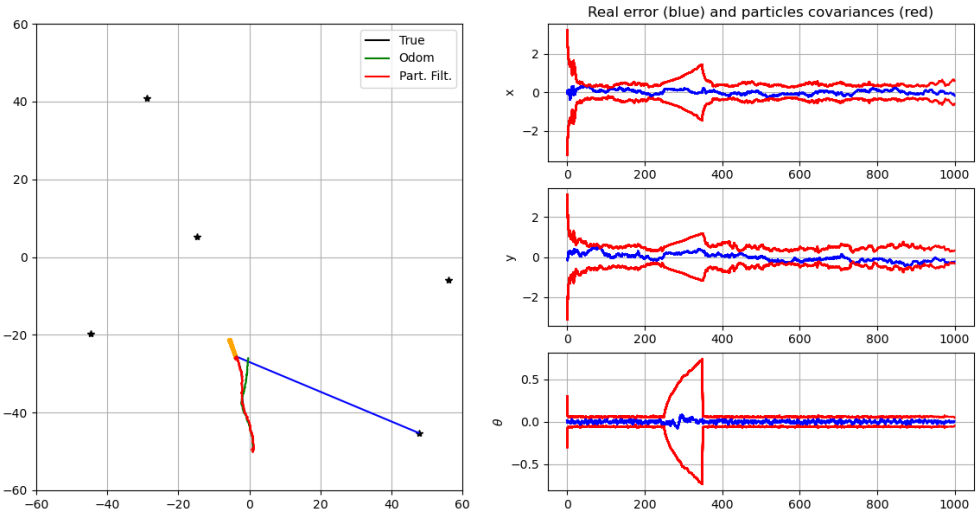
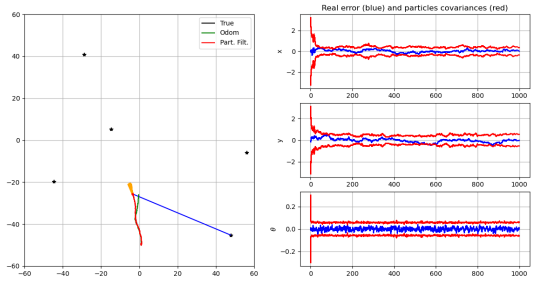


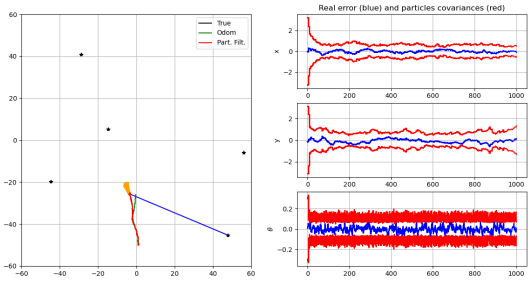
Figure 6: Filtre particulaire avec trou de mesures

7 Q7

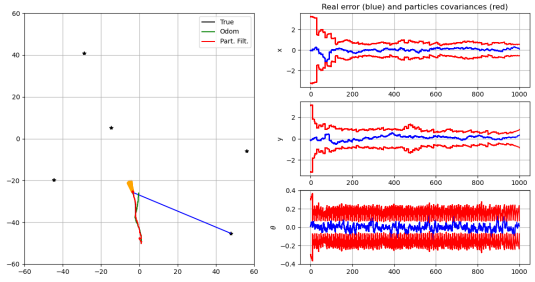
Les figures 7a à 7d correspondent à différentes trajectoires calculée par le filtre particulaire en fonction de la valeur de *dt_meas*, qui est l'intervalle de temps entre deux mesures de capteur. On remarque que la trajectoire corrigée est plus proche de la trajectoire réelle pour des valeurs de *dt_meas* petites, ce qui est logique car les poids des particules sont plus fréquemment mis à jour par les observations d'amères dans le terme d'innovation. Ceci permet de mieux pondérer le barycentre des particules, c'est-à-dire l'estimation de l'état du robot. De même, l'incertitude sur la position du robot tout au long de l'algorithme est plus faible quand *dt_meas* est petit. Enfin, on remarque que la convariance en augmente en x, y, et θ tant que le filtre ne peut pas corriger la trajectoire du robot à l'aide d'une mesure, puis diminue brusquement à l'acquisition de celle-ci. Ce phénomène est similaire à celui présenté en section 6, mais en périodique et de plus faible durée et intensité.



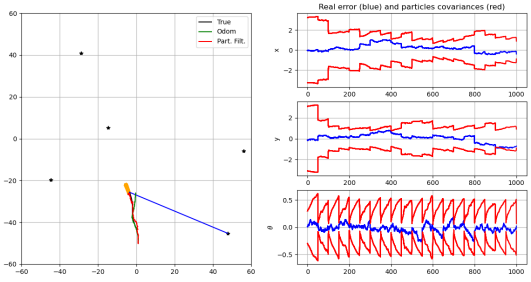
(a) *dt_meas* = 1



(b) *dt_meas* = 5



(c) *dt_meas* = 10

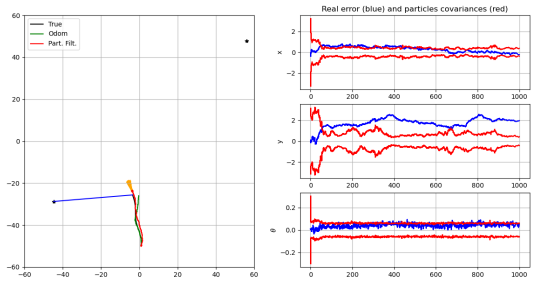


(d) *dt_meas* = 50

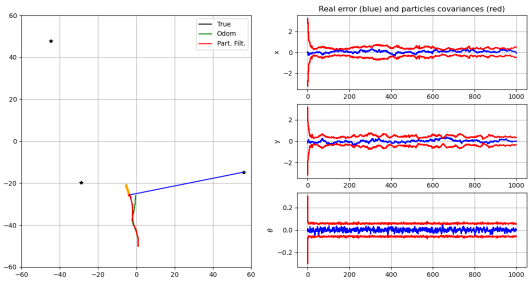
Figure 7: Filtre particulaire pour différentes valeurs de *dt_meas*

8 Q8

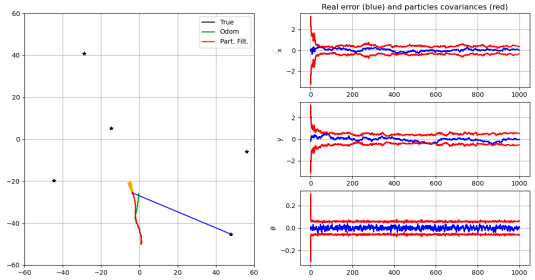
Les figures 8a à 8f correspondent à différentes exécutions du programme pour une quantité variable d'amères sur la carte. Seul le passage de 2 à 3 amères semble sensiblement impactant, réduisant drastiquement les erreurs en position et en rotation sur l'état prédit du robot. En augmentant le nombre d'amères, on observe plus de diminutions brusques de covariance, ce qui est dû à des observations d'amères de différentes positions, corrigeant plus efficacement la position du robot que si les amères étaient tous au même endroit. Cependant, la courbe de tendance de la covariance reste sensiblement la même à partir de 3 amères. On garde donc la valeur par défaut de 5 amères afin d'assurer une répartition suffisamment homogène des amères sur la carte, sans pour autant qu'il y en ait trop. À l'instar du filtre de Kalman, l'observation d'un amère à la fois (pour toutes les particules) rend l'augmentation du nombre d'amères moins intéressant que si un grand nombre d'amères pouvait être observé simultanément, mais cela réduit le temps de calcul de l'étape de prédiction du filtre ainsi que la taille des matrices et vecteurs avec lesquelles le filtre travaille.



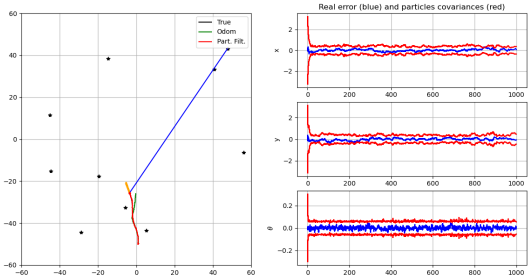
(a) 2 amères



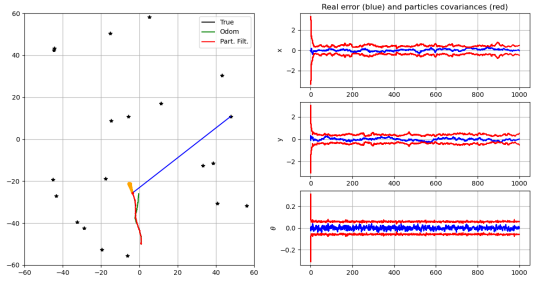
(b) 3 amères



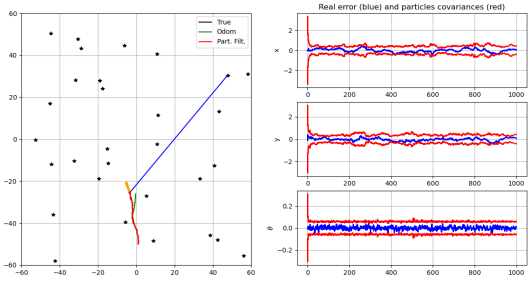
(c) 5 amères



(d) 10 amères



(e) 20 amères



(f) 30 amères

Figure 8: Filtre particulaire pour un nombre variable d'amères