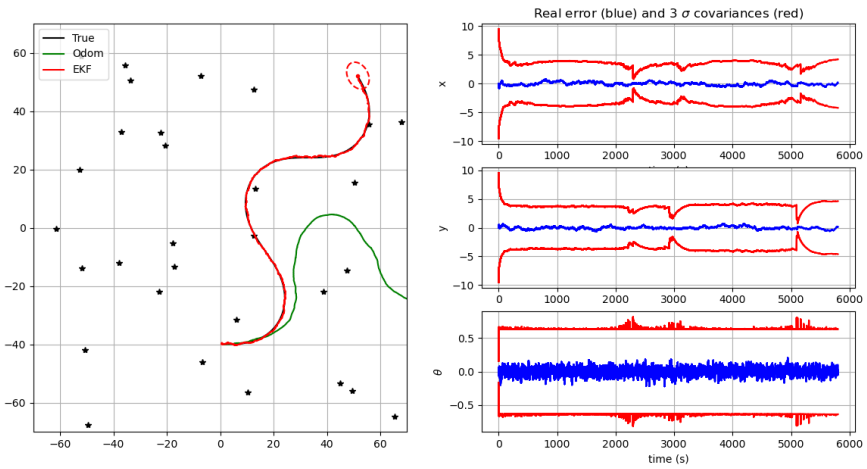
TP2 : Extended Kalman Filter

ROB312 - Navigation pour les systèmes autonomes

Bastien HUBERT



ENSTA Paris - octobre 2022

1 Q1

Le code ci-dessous correspond au code Python d'un filtre de Kalman étendu 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).

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

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.

```
# ---- Kalman 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 = [[sqrt( (xP - x)**2 + (yP - y)**2 )],
         [atan2(yP - y, xP - x) - theta]]

    return np.array(z)
```

1.3 Calculs des jacobiens

Cette partie correspond aux fonctions calculant les jacobiens des fonctions f et h définies plus haut.

```
# ---- Kalman Filter: Jacobian functions to be completed ----

# h(x) Jacobian wrt x
```

```
def get_obs_jac(xPred, iFeature, Map):
    # xPred: predicted state
    # iFeature: observed amer index
    # Map: map of all amers

    [x, y, theta] = xPred[:, 0]
    [xP, yP] = Map[:, iFeature]

    invDist = 1/sqrt((xP - x)**2 + (yP - y)**2)
    invDist2 = invDist * invDist

    jH = [[-invDist * (xP - x), -invDist * (yP - y), 0],
          [invDist2 * (yP - y), -invDist2 * (xP - x), -1]]

    return np.array(jH)

# f(x,u) Jacobian wrt x
def F(x, u, dt_pred):
    # x: estimated state (x, y, heading)
    # u: control input (Vx, Vy, angular rate)
    # dt_pred: time step

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

    df1_dtheta = (-vx * sin(theta_avant) - vy * cos(theta_avant)) * dt_pred
    df2_dtheta = (vx * cos(theta_avant) - vy * sin(theta_avant)) * dt_pred

    Jac = [[1, 0, df1_dtheta],
           [0, 1, df2_dtheta],
           [0, 0, 1]]

    return np.array(Jac)

# f(x,u) Jacobian wrt w (noise on the control input u)
def G(x, u, dt_pred):
    # x: estimated state (x, y, heading) in ground frame
    # u: control input (Vx, Vy, angular rate) in robot frame
    # dt_pred: time step for prediction

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

```
    Jac = dt_pred * [[cos(theta_avant), -sin(theta_avant), 0],
                     [sin(theta_avant), cos(theta_avant) , 0],
                     [0, 0, 1]]

    return np.array(Jac)
```

1.4 Fonctions auxiliaires

Cette partie correspond à des fonctions auxiliaires pour le calcul et l'affichage du filtre.

```
# ---- Utils functions ----
# Display error ellipses
def plot_covariance_ellipse(xEst, PEst, axes, lineType):
    """
    Plot one covariance ellipse from covariance matrix
    """

    Pxy = PEst[0:2, 0:2]
    eigval, eigvec = np.linalg.eig(Pxy)

    if eigval[0] >= eigval[1]:
        bigind = 0
        smallind = 1
    else:
        bigind = 1
        smallind = 0

    if eigval[smallind] < 0:
        print('Pb with Pxy :\n', Pxy)
        exit()

    t = np.arange(0, 2 * pi + 0.1, 0.1)
    a = sqrt(eigval[bigind])
    b = sqrt(eigval[smallind])
    x = [3 * a * cos(it) for it in t]
    y = [3 * b * sin(it) for it in t]
    angle = atan2(eigvec[bigind, 1], eigvec[bigind, 0])
    rot = np.array([[cos(angle), sin(angle)],
                    [-sin(angle), cos(angle)]])
    fx = rot @ (np.array([x, y]))
    px = np.array(fx[0, :] + xEst[0, 0]).flatten()
    py = np.array(fx[1, :] + xEst[1, 0]).flatten()
    axes.plot(px, py, lineType)
```

```
# 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
```

1.5 Programme principal

Cette partie correspond au filtre de Kalman étendu 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
# ================================================================

# Init displays
show_animation = True
save = False

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)
```

```
# ----- General variables -----

# Simulation time
Tf = 6000         # final time (s)
dt_pred = 1       # Time between two dynamical predictions (s)
dt_meas = 1       # Time between two measurement updates (s)

# Location of landmarks
Map = 140*(np.random.rand(2, 30) - 1/2)

# True covariance of errors used for simulating robot movements
QTrue = np.diag([0.01, 0.01, 1*pi/180]) ** 2
RTrue = np.diag([5.0, 6*pi/180]) ** 2

# Modeled errors used in the Kalman filter process
QEst = 100*np.eye(3, 3) @ QTrue
REst = 10*np.eye(2, 2) @ RTrue

# initial conditions
xTrue = np.array([[1, -40, -pi/2]]).T
xOdom = xTrue
xEst = xTrue
PEst = 10 * np.diag([1, 1, (1*pi/180)**2])

# Init history matrixes
hxEst = xEst
hxTrue = xTrue
hxOdom = xOdom
hxError = np.abs(xEst-xTrue)  # pose error
hxVar = np.sqrt(np.diag(PEst).reshape(3, 1))  # state std dev
htime = [0]

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

# Temporal loop
for k in range(1, simulation.nSteps):

    # Simulate robot motion
    simulation.simulate_world(k)

    # Get odometry measurements
```

```
    xOdom, u = simulation.get_odometry(k)

    # Kalman predictionl
    xPred = motion_model(xEst, u, dt_pred)                    # ^x_k|k-1 = f(^x_k-1, u_k)
    F_mat= F(xEst, u, dt_pred)                                # F = df/dx(^x_k)
    G_mat = G(xEst, u, dt_pred)                               # G = df/du(^x_k)
    PPred = F_mat @ PEst @ F_mat.T + G_mat @ QEst @ G_mat.T # ^P_k|k-1 = F_k * ^P_k-1 * F_k^T

    # Get random landmark observation
    [z, iFeature] = simulation.get_observation(k)

    if z is not None:
        # Predict observation
        zPred = observation_model(xPred, iFeature, Map)       # = h(^x_k|k-1)

        # get observation Jacobian
        H = get_obs_jac(xPred, iFeature, Map)                 # H_k = dh/dx(^x_k|k-1)

        # compute Kalman gain - with dir and distance
        Innov = z - zPred                                     # Innov_k = y_k - h(^x_k|k-1)
        Innov[1, 0] = angle_wrap(Innov[1, 0])
        S = REst + H @ PPred @ H.T                            # S_k = R_k + H_k * ^P_k|k-1 * H^
        K = PPred @ H.T @ np.linalg.inv(S)                    # K_k = ^P_k|k-1 * H_k^T * S_k^-1

        # Compute Kalman gain to use only distance
#         Innov = #.....                 # observation error (innovation) # TODO
#         H = #..... # TODO
#         S = #..... # TODO
#         K = #..... # TODO

        # Compute Kalman gain to use only direction
#         Innov = #.....                 # observation error (innovation) # TODO
#         Innov[1, 0] = angle_wrap(Innov[1, 0]) # TODO
#         H = #.....                     # observation error (innovation) # TODO
#         S = #..... # TODO
#         K = #..... # TODO

        # perform kalman update
        xEst = xPred + K @ Innov                              # ^x_k = ^x_k|k-1 + K_k * Innov
        xEst[2, 0] = angle_wrap(xEst[2, 0])
        PEst = ( np.eye( 3 ) - K @ H ) @ PPred                # ^P_k = (I_d - K_k * H_k ) * ^P_
        PEst = 0.5 * (PEst + PEst.T)  # ensure symetry
    else:
```

```
        # there was no observation available
        xEst = xPred
        PEst = PPred

#     print("xPred\n", xPred)
#     print("PPred\n", PPred)
#     print("F_mat\n", F_mat)
#     print("G_mat\n", G_mat)
#     print("H\n", H)
#     print("S\n", S)
#     print("K\n", K)
#     print("xEst\n", xEst)
#     print("PEst\n", PEst)

    # 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))
    hxVar = np.hstack((hxVar, np.sqrt(np.diag(PEst).reshape(3, 1))))
    htime.append(k*simulation.dt_pred)

    # plot every 15 updates
    if show_animation and k*simulation.dt_pred % 200 == 0:
        print(k)

        # 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()

        times = np.stack(htime)

        # Plot true landmark and trajectory
        ax1.plot(Map[0, :], Map[1, :], "*k")
        ax1.plot(hxTrue[0, :], hxTrue[1, :], "-k", label="True")

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

```
        # Plot estimated trajectory an pose covariance
        ax1.plot(hxEst[0, :], hxEst[1, :], "-r", label="EKF")
        ax1.plot(xEst[0], xEst[1], ".r")
        plot_covariance_ellipse(xEst,
                                PEst, ax1, "--r")

        ax1.axis([-70, 70, -70, 70])
        ax1.grid(True)
        ax1.legend()

        # plot errors curves
        ax3.plot(times, hxError[0, :], 'b')
        ax3.plot(times, 3.0 * hxVar[0, :], 'r')
        ax3.plot(times, -3.0 * hxVar[0, :], 'r')
        ax3.grid(True)
        ax3.set_ylabel('x')
        ax3.set_xlabel('time (s)')
        ax3.set_title('Real error (blue) and 3 $\sigma$ covariances (red)')

        ax4.plot(times, hxError[1, :], 'b')
        ax4.plot(times, 3.0 * hxVar[1, :], 'r')
        ax4.plot(times, -3.0 * hxVar[1, :], 'r')
        ax4.grid(True)
        ax4.set_ylabel('y')
        ax5.set_xlabel('time (s)')

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

        if save: plt.savefig(r'outputs/EKF_' + str(k) + '.png')
#         plt.pause(0.001)

plt.savefig('EKFLocalization.png')
print("Press Q in figure to finish...")
plt.show()
```

2 Q2

Le code complété est présenté aux sections 1.2, 1.3 et 1.5. L'exécution de ce code conduit à la figure 1 :

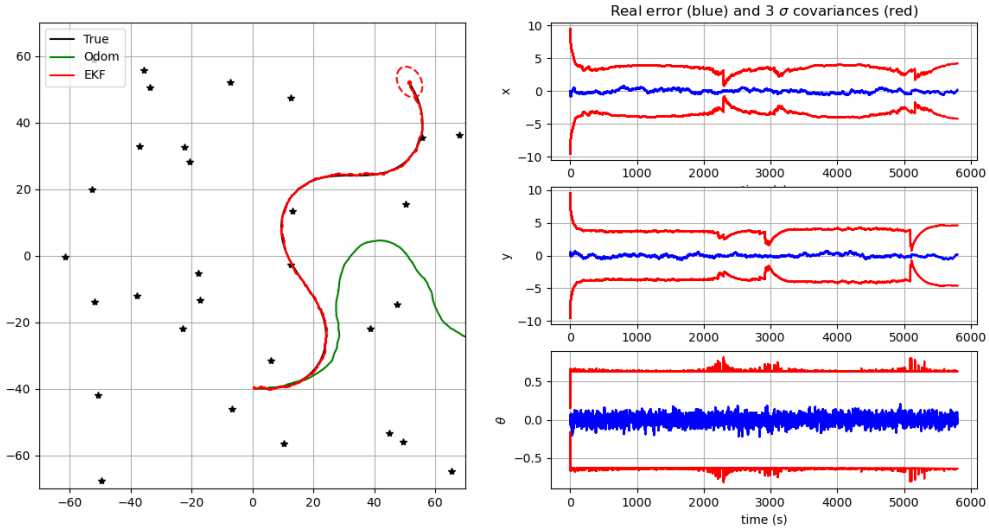
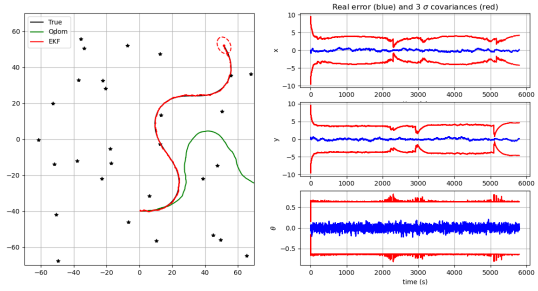


Figure 1: Filtre de Kalman étendu

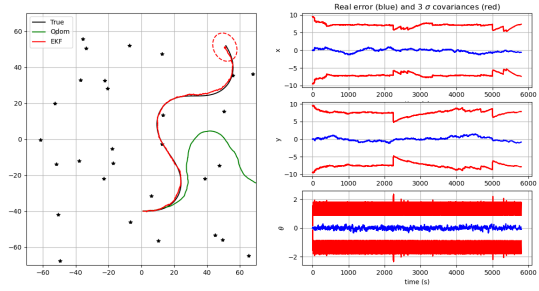
On remarque que la trajectoire corrigée par le filtre est très proche de la trajectoire réelle malgré la dérive importante de l'odométrie du robot. De plus, l'incertitude sur la position réduit très vite lors des premières itérations du filtre, puis se stabilise en régime permanent.

3 Q3

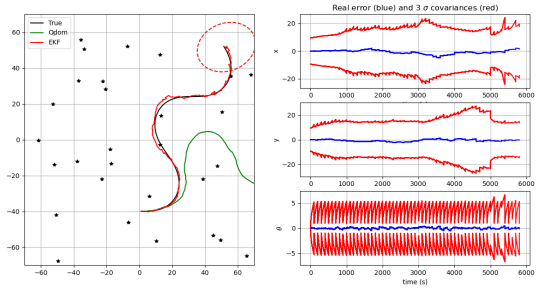
Les figures 2a à 2d correspondent à différentes trajectoires calculée par le filtre de Kalman en fonction de la valeur de *dt_meas*, qui est l'intervalle de temps entre deux mesures de capteur. On remarque que la trajectoire corrigée est bien plus proche de la trajectoire réelle pour des valeurs de *dt_meas* petites, ce qui est logique car celle-ci est alors fréquemment corrigée par les comparaisons avec les amères. 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.



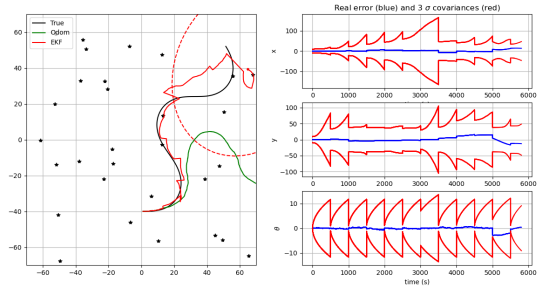
(a) $dt\_meas = 1$



(b) $dt\_meas = 10$



(c) $dt\_meas = 100$

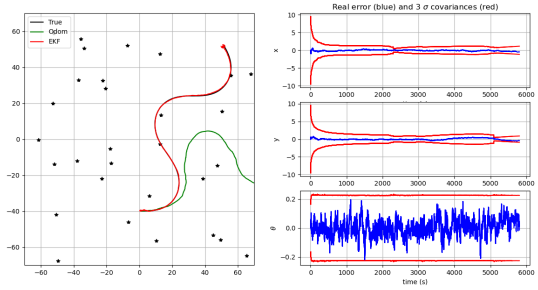


(d) $dt\_meas = 500$

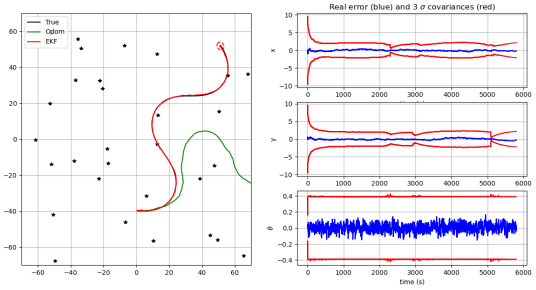
Figure 2: Filtre de Kalman pour différentes valeurs de *dt_meas*

4 Q4

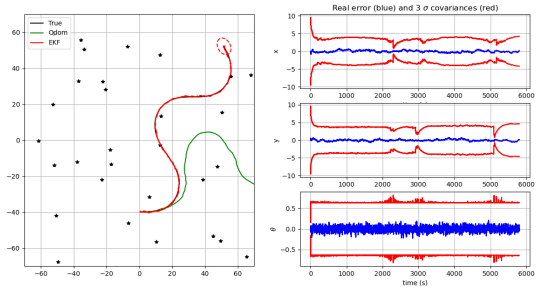
Les figures 3a à 3d correspondent à différentes trajectoires calculée par le filtre de Kalman en fonction de la valeur de $QEst$, qui traduit le bruit dynamique du filtre, c'est-à-dire la confiance que l'on a dans l'estimation de la position du robot par son odométrie. Un coefficient multiplicatif important 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.



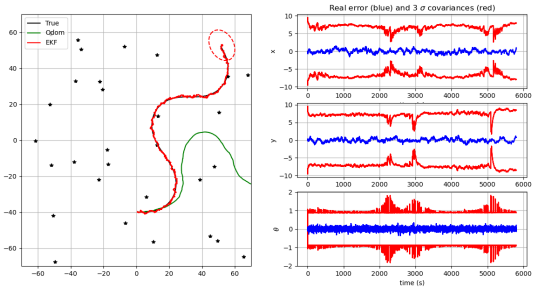
(a) $QEst = 1 * np.eye(3,3) \,@\, QTrue$



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



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

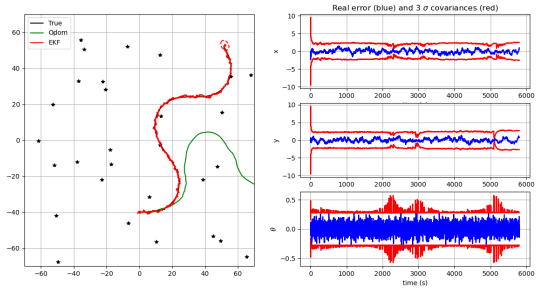


(d) $QEst = 1000 * np.eye(3,3) \,@\, QTrue$

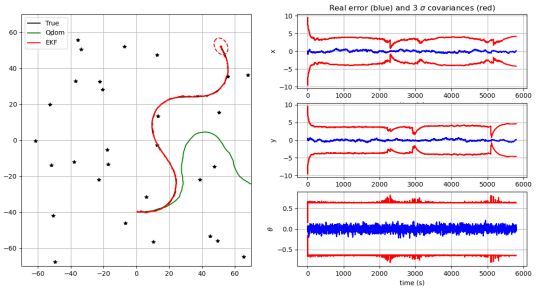
Figure 3: Filtre de Kalman pour différentes valeurs de $QEst$

5 Q5

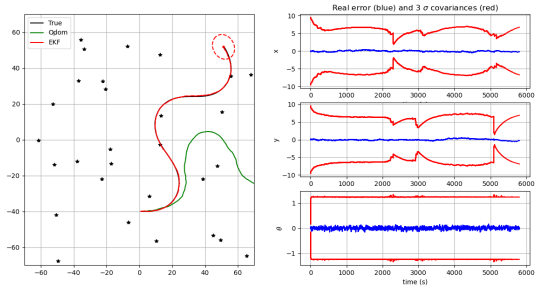
Les figures 4a à 4d correspondent à différentes trajectoires calculée par le filtre de Kalman en fonction de la valeur de $REst$, qui traduit le bruit de mesure du filtre, c'est-à-dire la confiance que l'on a dans la mesure de la position du robot par rapport à un amère. Un coefficient multiplicatif 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, et une covariance plus importante et réduisant moins souvent. On rappelle que les réductions brusques de covariance correspondent aux acquisitions de mesures qui rendent beaucoup plus précis la position du robot, comme la comparaison à un amère proche du robot.



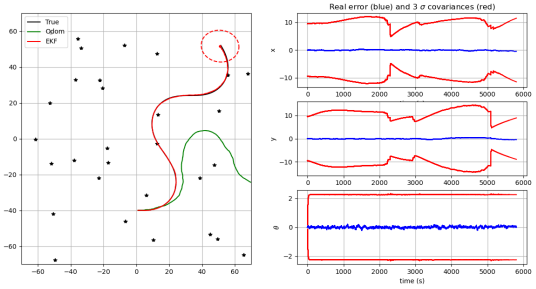
(a) $REst = 1 * np.eye(2,2)$ @ $RTrue$



(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 4: Filtre de Kalman pour différentes valeurs de $QEst$

6 Q6

On simule un trou de mesure en rajoutant la condition $notValidCondition = True$ pour $k \in [\![2500, 3500]\!]$ dans le code de *get_observation* de la section 1.1. L'exécution du programme avec cette perte des mesures est illustré par la figure 5. On remarque la ressemblance avec un unique pic de la figure 2d: 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.

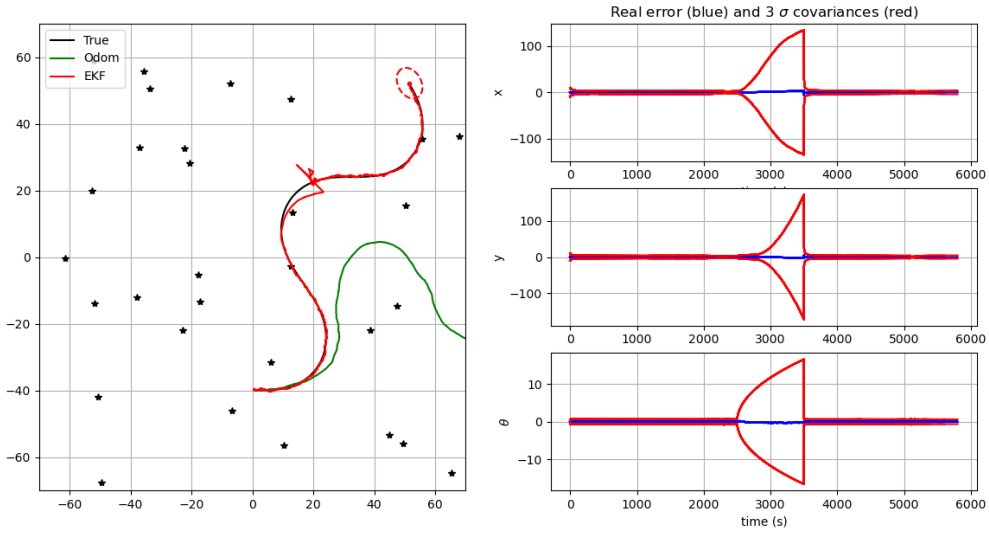
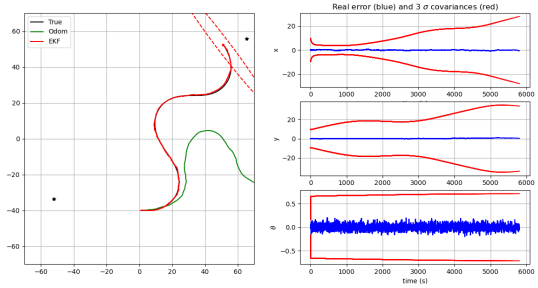


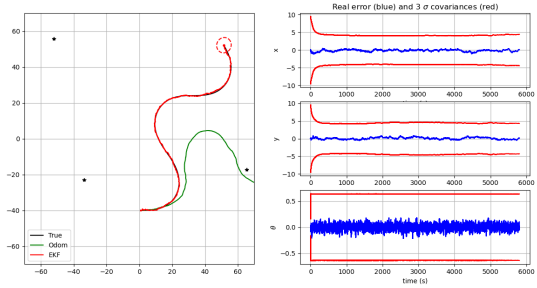
Figure 5: Filtre de Kalman avec trou de mesures

7 Q7

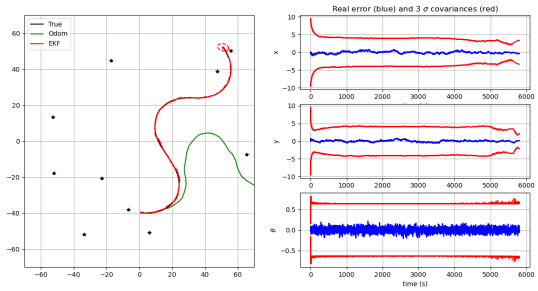
Les figures 6a à 6f correspondent à différentes exécutions du programme pour une quantité variable d'amères sur la carte. Seul le passage de 2 à 3 amères semble sensiblement impactant, passant d'une ellipse de covariance est très aplatie le long de la direction orthogonale à celle reliant les deux amères, à une ellipse beaucoup plus proche d'un cercle à partir de trois amères. 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 30 amères afin d'assurer une répartition suffisamment homogène des amères sur la carte, sans pour autant qu'il y en ait trop.



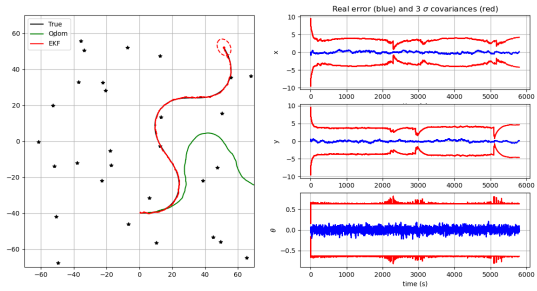
(a) 2 amères



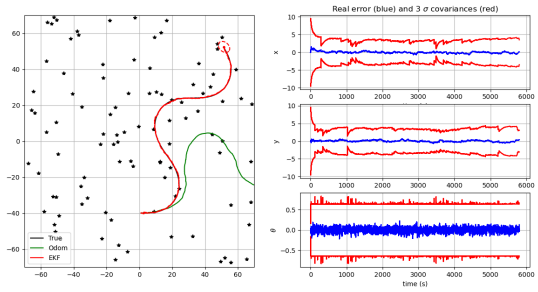
(b) 3 amères



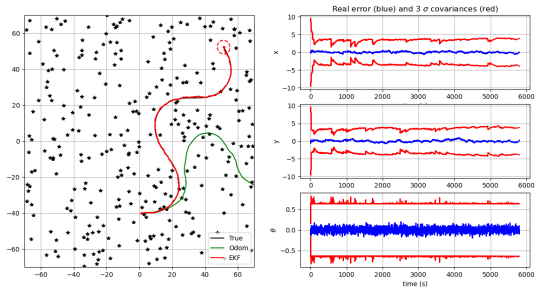
(c) 10 amères



(d) 30 amères



(e) 100 amères



(f) 300 amères

Figure 6: Filtre de Kalman pour un nombre variable d'amères