6.1 Mapping EFK - The Nirvana shopping mall

The managers behind the **Nirvana shopping mall**, in the scope of their *experiencing the future* plan, has contacted **UMA-MR** (our brand-new mobile robotics company at UMA) looking for mobile robots able to guide their visitors between different points of interest in their facilities, like information points, the entrance to relevant (paying well) shops, rescue points, etc.

NIRVANA SHOPPING MALL

EXPERIENCING THE FUTURE | | WITH YOU

These managers have placed identifying marks close to these points of interest, but do not know their exact location in the mall. The image below shows an example of this:

Our mission as engineers at *UMA-MR* is to build a map of the mall containing such points, so the robot can operate within it. We are going to use for that an **Extended Kalman Filter (EKF)**.

Fortunately, our company has developed a system able to provide the exact location of our robots at each time instant, which are also equipped with a range-and-bearing nosiy sensor able to detect the identifying marks and take measurements to them.

6.2.1 Formalizing the problem

Since we are going to build a map of N landmarks, the position of those landmarks in the map m are the random variables to be estimated. In this way, the state vector in the mapping case is defined as:

$$m = [m_1, m_2, \cdots, m_N] = [x_1, y_1, \underbrace{x_2, y_2}_{\substack{\text{Position of} \\ \text{landmark 2}}}, \ldots, x_N, y_N]^T, \quad len(m) = 2N$$

In other words, we pursuit the estimation of the probability distribution:

$$p(m|z_{1:t}, x_{1:t})$$

being $z_{1:t}$ the sensor measurements taken until time instant t, and $x_{1:t}$ the robot poses from which those measurements were acquired. Recall that a sensor measurement is related to the pose x and the map m by means of the **observation function**:

$$z = h(x, m) + e \qquad e \sim N(0, Q)$$

Two assumptions are made when building maps using EFK.

Assumption 1: each landmark position is estimated independently

For simplifying the problem, it is usual to assume that the estimation of the position of the landmarks is independent one to another, only depending each one on its observations, so:

$$p(m_i|z_{1:t}, x_{1:t}) = p(m_i|z^i_{1:t}, x_{1:t})$$

which results in the following simplification:

$$p(m|z_{1:t}, x_{1:t}) = p(m_1, m_2, \cdots, m_N|z_{1:t}, x_{1:t}) = \prod_{k=1}^{N} p(m_i|z^i_{1:t}, x_{1:t})$$

As a consequence of this, the estimation of, for example, 3 landmarks simultaneously is the same as using three concurrent and independent EKFs for estimating them.

Assumption 2: the map is static

Unlike the localization case, where the robot pose changed over time, in this case it is assumed that the map is static, that is, the landmarks are still. This means that in the state transition model $m_k = m_{k-1}$, that is:

$$m_t = A_t m_{t-1} + B_t u_t + \epsilon_t \quad (A = I, u = 0, \epsilon = 0)$$

So good news here!, there is no need for a prediction step in the EKF, we only have to model the correction (update) one.

6.2.2 Developing the EKF filter for mapping the mall

In [ ]:
```
%matplotlib widget

import sys
import time

import numpy as np
```

```python
from numpy import random
from scipy import linalg
import matplotlib
import matplotlib.pyplot as plt

import sys
sys.path.append("..")
from utils.tcomp import tcomp
from utils.AngleWrap import AngleWrap
from utils.DrawRobot import DrawRobot
from utils.PlotEllipse import PlotEllipse
from utils.Jacobians import J2
from utils.unit6.MapCanvas import MapCanvas
```

The provided robot

Take a look at the following `EFKMappingRobot()` class, which is already provided by your colleagues at **UMA-MR**, modeling **a robot equipped with a range and bearing sensor**, which is able to keep the following information:

- `true_pose` : The exact pose of the robot in the environment, which is perfectly known.
- `Q` : Uncertainty of the range and bearing sensor, which has the form: $\Sigma_{r\theta} = \begin{bmatrix} \sigma_r^2 & 0 \\ 0 & \sigma_\theta^2 \end{bmatrix}$
- `xEst` : Vector with the estimated state, in this case the position of the M observed landmarks: $[x_1, y_1, x_2, y_2, \ldots, x_M, y_M]^T$. Its size changes over time with the detection of previously unobserved landmarks.
- `PEst` : uncertainty associated with those predictions, with size ($M \times 2$,$M \times 2$). Its size also changes over time.
- `MappedLandmarks` : A vector with length equal to the number of landmarks in the map (N), which elements can take the following values:
  - `-1` if the landmark with that index has not been seen yet.
  - `idx_in_xEst` : an odd number indicating the position of that landmark in `xEst` . For example, if during the robot operation in a map with 5 landmarks, it first detect the landmark with id 2 (ids starts at 0), and later the one with id 4, the content of this vector would be `MappedLandmarks=[-1,-1,1,-1,3]` .

and to perform the following actions:

- `step()` : Performs a motion command, without noise.
- `observe()` : Returns a range and bearing measurement (in polars) to a given landmark in the map.
- `get_random_observation()` : Returns a range and bearing measurement (in polars) to a random landmark.

In [ ]:
```python
class EFKMappingRobot():
    def __init__(self, true_pose, sigma_r, sigma_theta, n_features):
        # Robot description
        self.true_pose = true_pose
        self.Q = np.diag([sigma_r, sigma_theta])**2

        # Map -- Initially empty
        self.xEst = np.empty((0, 0))
        self.PEst = np.empty((0, 0))
        self.QEst = 1.0*self.Q

        self.MappedLandmarks = -1*np.ones((n_features,1), int)
    def step(self, u):
        self.true_pose = tcomp(self.true_pose, u)

    def observe(self, idx, world, noisy=True):
        """ Generate a observation of a feature in our world

            Args:
                world: Complete map of all landmarks in the world
                idx: Landmark to observe (index in world matrix)
                noisy: Add noise to z, prorportional to self.Q

            Returns:
                z: One range and bearing observation
        """

        z = np.empty((2, 1))
        delta = world[:, [idx]] - self.true_pose[0:2, :]

        # Range
        z[0,:] = np.sqrt(np.sum(delta**2))
        # Bearing
        z[1,:] = np.arctan2(delta[1, 0],delta[0, 0])-self.true_pose[2, 0]
        z[1,:] = AngleWrap(z[1, :])

        if noisy:
            z = z+np.sqrt(self.Q)@random.randn(2,1)

        return z

    def get_random_observation(self, world, noisy=True):
        iLandmarks = world.shape[1]
        iLandmark = random.randint(iLandmarks)
        z = self.observe(iLandmark, world, noisy)
        return z, [iLandmark]
```

The prediction step

As commented, the map is considered static, so the prediction step is reduced to consider as the predicted landmarks' positions the ones estimated in the previous step. The same holds for the predicted uncertainty, so this steps results in something like this:

```
def ExtendedKalmanFilter$(m_{t-1}, \Sigma_{t-1}, z_t)$ :
    Prediction.
    $\bar{m}_t = m_{t-1}$                    (1. Map prediction)
    $\bar{\Sigma}_t = \Sigma_{t-1}$          (2. Uncertainty of prediction)
```

ASSIGNMENT 1: Implementing the naive prediction step

You are tasked to implement the previous behavior in the following function, which performs the prediction step.

```python
In [ ]: def prediction_step(robot: EFKMappingRobot):
            """ Performs the prediction step of the EKF algorithm for mapping
                    robot: Robot base (contains state map: xEst, PEst)

                Returns: Nothing. But it modifies the state in robot
                    xPred: Predicted position of the landmarks
                    PPred: Predicted uncertainty of the landmarks positions
            """
```

```
    # We assume that the map is static
    xPred = robot.xEst
    PPred = robot.PEst

    return xPred, PPred
```

You can **test your function** with the next code:

In [ ]:
```
# TRY IT!
xVehicleTrue = np.vstack([0.5, 0.7, 0]) # We know the exact robot pose at any moment
robot = EFKMappingRobot(xVehicleTrue, 1, 0.8, 1)
robot.xEst = np.vstack([.5, .7])
robot.PEst = np.diag([1.32, 0.8])

[xPred,PPred] = prediction_step(robot)

print('>> Previous estimation <<')
print('xEst:\n' + str(robot.xEst))
print('PEst:\n' + str(robot.PEst) + '\n')
print('>> Prediction <<')
print('xPred:\n' + str(xPred))
print('PPred:\n' + str(PPred))
```

```
>> Previous estimation <<
xEst:
[[0.5]
 [0.7]]
PEst:
[[1.32 0.  ]
 [0.   0.8 ]]

>> Prediction <<
xPred:
[[0.5]
 [0.7]]
PPred:
[[1.32 0.  ]
 [0.   0.8 ]]
```

Expected output:

```
xPred:
[[0.5]
 [0.7]]
PPred:
[[1.32 0.  ]
 [0.   0.8 ]]
```

Observing a landmark for first time

When the sensor onboard the robot detects a landmark for the first time, there is no need to do the EKF update step (indeed, since there is not previously knowledge about the landmark, there is nothing to update). Instead, we have to properly modify 1) the vector of estimated landmark positions, and 2) their associated uncertainties, to accommodate this new information:

1. **Modifying the state vector**: Insert the position of the new observed landmark $[x_{M+1}, y_{M+1}]$, using the sensor measurement $z_k = [r_k, \theta_k]$, at the end of the vector containing the estimated positions `xEst`, so:

$$xEst = [x_1, y_y, \cdots, x_M, y_M, x_{M+1}, y_{M+1}]$$

Since the measurment is provided in polar coordinates in the robot local frame, we have to convert them first to cartensians and then to the world frame using the robot pose $[x_v, y_v, \theta_v]'$. The function in charge of doing so can be defined as:

$$f(x_v, z_k) = \begin{bmatrix} x_{M+1} \\ y_{M+1} \end{bmatrix} = \begin{bmatrix} x_v \\ y_v \end{bmatrix} + r_k \begin{bmatrix} cos\alpha_k \\ sin\alpha_k \end{bmatrix}, \quad \alpha_k = \theta_k + \theta_v$$

2. **Extending the covariance matrix**. In order to acomodate the uncertainty regarding the position of the new landmark, we have to extend the covariance matrix in the following way:

$$PEst = \begin{bmatrix} [\Sigma^1_{xy}]_{2\times2} & \cdots & 0_{2\times2} \\ \vdots & \ddots & \vdots \\ 0_{2\times2} & \cdots & [\Sigma^{M+1}_{xy}]_{2\times2} \end{bmatrix}_{2n\times2n}$$

Notice that the covariance Σ^{M+1}_{xy} stands for the uncertainty in the measurement expressed in the world cartesian coordinates, retrieved by:

$$\Sigma^{M+1}_{xy} = J\Sigma^{M+1}_{r\theta}J^T$$

being $\Sigma^{M+1}_{r\theta}$ the uncertainty characterizing the sensor measurements (`QEst` in our code), and J (`jGz` in our code) the jacobian of the function $f(x_v, z_k)$ that expresses the measurement in global coordinates, which is:

$$J = \begin{bmatrix} \partial x/\partial r & \partial x/\partial\theta \\ \partial y/\partial r & \partial y/\partial\theta \end{bmatrix} = \begin{bmatrix} cos\alpha & -rsin\alpha \\ sin\alpha & rcos\alpha \end{bmatrix}$$

Notice that this jacobian is the result of concatenating both, the jacobian of the composition of a pose and a landmark, and the jacobian of the function transforming polar into cartesian coordinates. It is quite similar to the one just transforming polar into cartesian, but in this case the angle α is the sum of the robot orientation and the measured angle (bearing).

ASSIGNMENT 2: Incorporating a landmark detected for first time

Your work here is to:

- Complete the `get_new_landmark_jacobians()` to compute the jacobian J.
- Complete the `incorporate_new_landmark()` method to modify the state vector `xEst` and the convariance matrix `PEst` as explained above. We will make use of the `linalg.block_diag()` function at this point.

In [ ]:
```
def get_new_landmark_jacobians(Xv, z):
    """ Calculate the jacobian for transforming an observation to the world frame

        Args:
            Xv: True pose of our robot
            z: Observation of a landmark. In polar coordinates from the p.o.v. of our rob
        Returns:
            2x2 matrix containing the corresponding jacobian.
    """
    r,a = z[0,0], z[1,0] + Xv[2,0]
    c,s = np.cos(a), np.sin(a)
    jGz = np.array([
        [c,-r*s],
        [s,r*c]
    ])
    return jGz
```

Is it working properly? **Try it!**

```
In [ ]: # TRY IT!
        np.set_printoptions(threshold=np.inf, precision=8, suppress=True)
        z = np.vstack([1.2,0.35])
        Xv = np.vstack([2, 2.1, 0])
        jGz = get_new_landmark_jacobians(Xv, z)
        print('jGz:\n' + str(jGz))
```

```
jGz:
[[ 0.93937271 -0.41147737]
 [ 0.34289781  1.12724726]]
```

Expected output:

```
jGz:
[[ 0.93937271 -0.41147737]
 [ 0.34289781  1.12724726]]
```

```
In [ ]: def incorporate_new_landmark(robot: EFKMappingRobot, z, iLandmark, xPred, PPred):
            """ Incorporates the information relative to a new ladmark to our system
                robot: Robot base (contains state map: xEst, PEst)
                z: Observation of a landmark
                iLandmark: Index of z in the world map
                xPred: Predicted map
                PPred: Uncertainty of the prediction

                Returns: Nothing. But it modifies the state in robot
            """
            # This is a new feature, so add it to the map

            # The observation is in the local frame of the robot, it has to
            # be translated to the global frame
            r, a = z[0,0], z[1,0] + Xv[2,0]
            c, s = np.cos(a), np.sin(a)
            xLandmark = robot.true_pose[0:2] + r * np.vstack([c,s])

            # Add it to the current state
            nStates = xPred.size

            if nStates == 0:
                robot.xEst = xLandmark
            else:
                robot.xEst = np.vstack([robot.xEst, xLandmark]) #Each new feature two new rows

            # Compute the jacobian
            jGz = get_new_landmark_jacobians(robot.true_pose, z) #Dimension 2x2

            # Build a matrix M incorporating the jacobian to multiply the extendend PEst matrix b
            if nStates != 0:
                # note we don't use jacobian w.r.t vehicle since the pose doesn’t have uncertaint
                M = np.vstack([
                        np.hstack([np.eye(nStates), np.zeros((nStates, 2))]),
                        np.hstack([np.zeros((2, nStates)), jGz])
                    ])
            else:
                # First landmark observed!
                M = jGz

            robot.PEst = M@linalg.block_diag(robot.PEst, robot.QEst)@M.T

            #This can also be done directly PEst = [PEst,zeros(nStates,2);
                                                    # zeros(2,nStates),
                                                    # jGz*QEst*jGz']

            #remember this landmark as being mapped: we store its ID for the state vector
            robot.MappedLandmarks[iLandmark] = robot.xEst.size-2 #Always an odd number
```

The correction (update) step

Once a landmark has been detected and its provided information (location and uncertainty) has been properly incorporated to our mapping system, such an information can be updated with new measurements of such a landmark. For doing so, the EKF algorithm performs the following steps:

```
Correction.
$K_t = \bar{\Sigma}_t H_t^T (H_t \bar{\Sigma}_t H_t^T + Q_t)^{-1}$          (3. Kalman gain)
$m_t = \bar{m}_t + K_t(z_t - h(x_t, \bar{m}_t))$          (4. Map estimation)
$\Sigma_t = (I - K_t H_t)\bar{\Sigma}_t$          (5. Uncertainty of estimation)
return $m_t, \Sigma_t$
```

Notice that the map (landmark locations) is estimated according to the map estimation in the previous time step $t-1$, the *Kalman gain*, and the error (also called ***innovation***) between the observation taken by the sensor (z_t) and the one computed by the observation model given the robot pose and the predicted map, that is $z_t - h(x_t, \bar{m}_t)$.

In such equations, H (`jH` in our code) represents the jacobian of the observation model. The shape of such a jacobian is $(2, M \times 2)$, and has the following form:

$$H = \begin{bmatrix} 0 & 0 & \cdots & \underbrace{\begin{bmatrix} jHxl_{11} & jHxl_{12} \\ jHxl_{21} & jHxl_{22} \end{bmatrix}}_{\text{Jacobian for the observed landmark, } jHxl} & \cdots & 0 & 0 \\ 0 & 0 & \cdots & & \cdots & 0 & 0 \end{bmatrix}$$

$$jHxl = \begin{bmatrix} (x_l - x)/d & (y_l - y)/d \\ -(y_l - y)/d^2 & (x_l - x)/d^2 \end{bmatrix}$$

where $[x_l, y_l]$ is the position of the landmark, $[x, y]$ is the robot location, and $d = \sqrt{(x_l - x)^2 + (y_l - y)^2}$.

ASSIGNMENT 3: Updating the knowledge about an observed landmark

Your job at this point is:

- To implement the function `get_observation_jacobian()` returning the jacobian of the observed landmark, that is, $jHxl$.
- To complete the `update_step()` method that performs the update step of the EKF algorithm.

```
In [ ]: def get_observation_jacobian(xPred, xLandmark):
            """ Calculate the jacobian of the observation model.

                Needed to update a landmark we have already seen.
                Hint. Similar to the one described in unit 5 (Localization)

                Args:
                    xPred: True pose of our robot.
                    xLandmark: Estimated pose of a landmark in our map. World p.o.v in cartesian
                        Does not contain an angle.

                Return:
                    jHxl: 2x2 matrix containing the corresponding jacobian.
            """
            xdist = xLandmark[0, 0] - xPred[0, 0]
            ydist = xLandmark[1, 0] - xPred[1, 0]
            r = np.sqrt(xdist**2 + ydist**2)
            r2 = r**2
            jHxl = np.array([
                [xdist / r , ydist / r],
                [-ydist / r2 , xdist / r2]
            ])
            return jHxl
```

Let's see if your implementation is right:

```
In [ ]: # TRY IT!
        xLandmark = np.vstack([2.5,2])
        xPred = np.vstack([2, 2.1, 0])
        jHxl = get_observation_jacobian(xPred, xLandmark)
        print('jHxl:\n' + str(jHxl))
```

```
jHxl:
[[ 0.98058068 -0.19611614]
 [ 0.38461538  1.92307692]]
```

Expected output:

```
jHxl:
[[ 0.98058068 -0.19611614]
 [ 0.38461538  1.92307692]]
```

```
In [ ]: def update_step(robot, z, iLandmark, xPred, PPred):
            """ Performs the update step of EKF
                robot: Robot base (contains state map: xEst, PEst)
                z: Observation of a landmark
                iLandmark: Index of z in the world map
                xPred: Predicted map
                PPred: Uncertainty of the prediction

                Returns: Nothing. But it modifies the state in robot
            """
            # Find out where it is in state vector
            landmarkIndex = robot.MappedLandmarks[iLandmark[0], 0]

            # xLandmark is the current estimation of the position of the
            # landmard "FeatureIndex"
            xLandmark = xPred[landmarkIndex:landmarkIndex+2]

            # DONE Predicts the observation
            zPred = robot.observe(0, xLandmark, noisy=False) # Hint: use robot.observe function

            # Get observation Jacobians
            jHxf = get_observation_jacobian(robot.true_pose, xLandmark)
```

```
    # Fill in state jacobian
    # (the jacobian is zero except for the observed landmark)
    jH = np.zeros((2, xPred.size))
    jH[:, landmarkIndex:landmarkIndex+2] = jHxf

    #
    # Kalman update
    #
    Innov = z - zPred # Innovation
    Innov[1] = AngleWrap(Innov[1])
    S = jH @ PPred @ jH.T + np.diag(np.tile(np.diag(robot.QEst), xLandmark.shape[0] // 2
    K = PPred @ jH.T @linalg.inv(S) # Gain
    robot.xEst = xPred + K @ Innov
    robot.PEst = (np.eye(robot.PEst.shape[0]) - K @ jH) @ PPred
    #robot.PEst = PPred - K@S@K.T # Alternative way

    #ensure P remains symmetric
    robot.PEst = 0.5*(robot.PEst+robot.PEst.T)
```

ASSIGNMENT 4: Putting all together in the EKF algorithm

Now that you have implemented the building blocks of the EKF filter for mapping the **Nirvana shopping mall**, it is time to write a simple function `EKFMapping()` putting them together. For that, **you have to** call each method with the appropriate parameters.

In [ ]:
```
def EKFMapping(robot: EFKMappingRobot, z, iLandmark):
    """ EFK algorithm for mapping

            robot: Robot base (contains state map: xEst, PEst)
            z: Observation of a landmark
            iLandmark: Index of z in the world map

        Returns: Nothing. But it modifies the state in robot
    """

    # Do prediction step
    [xPred, PPred] = prediction_step(robot)

    # Check if feature observed is in map
    if robot.MappedLandmarks[iLandmark] > -1:
        update_step(robot,z,iLandmark,xPred,PPred)

    else:
        # This is a new feature, so add its information to the map
        incorporate_new_landmark(robot, z, iLandmark, xPred, PPred)
```

6.2.3 Testing the mapping system

Playing with one landmark

Let's consider that the mall has only one landmark to get things started (`nLandmarks=1`). The following function provides a demo where the robot is commanded to follow a squared trajectory while observing a landmark after each movement.

The **Nirvana** managers are curious about the state and dimensions of the variables storing the estimated positions `xEst` and their associated uncertainties `Pest` , so we show their content after each 5 iterations of the algorithm.

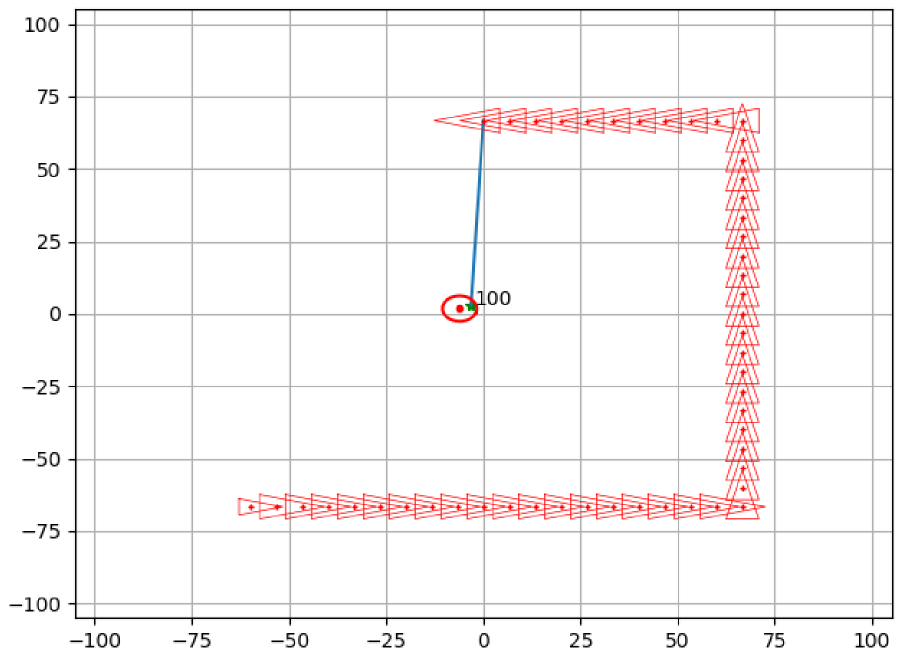

Fig. 1: Example run of the EKF algorithmn for mapping (only one landmark).
it shows the true pose (in red),
the real pose of the landmark (as a green star),
and the estimation from the EKF algorithm (pose and confidence ellipse).

In [ ]:
```
def demo_ekf_mapping(robot,
                     Map,
                     nLandmarks,
                     mode='non_stop',
                     logger=None,
                     nSteps=100, # Number of motions
                     turning= 40, # Number of motions before turning (square path)
                     print_each=2):

    %matplotlib widget
    if mode == 'step_by_step':
        matplotlib.use('TkAgg')

    # storing the number of times a landmark has been seen
    # also store the handler to the graphical info shown
    canvas = MapCanvas(nLandmarks)

    canvas.ax.plot(Map[0, :], Map[1, :], 'g*')
    hObsLine = canvas.ax.plot([0,0], [0,0], linestyle=':')

    # Control action
    u = np.zeros((3, 1))
    u[0] = (2.*MapSize/1.5)/turning
    u[1] = 0.

    log_xEst = []
    log_PEst = []

    # Start the Loop!
    for k in range(nSteps):
        #
        # Move the robot
        #
        u[2]=0.
        if k%turning == turning-1:
            u[2] = np.pi/2
        robot.step(u) # Perfectly known robot pose

        z, iLandmark = robot.get_random_observation(world=Map)

        # Update the "observedtimes" for the feature and plot the reading
        canvas.increment_observed_times(iLandmark)
        canvas.PlotNumberOfReadings(robot.true_pose, iLandmark, Map)

        EKFMapping(robot, z, iLandmark)

        # Store map evolution each 5 steps
        if not k%5:
            log_xEst.append(robot.xEst)
            log_PEst.append(robot.PEst)

        # Log important values
        if logger is not None:
            logger.log(k, robot, Map)

        # Drawings
        if k%print_each == print_each-1:
            DrawRobot(canvas.fig, canvas.ax,robot.true_pose, 'r')#plot(xVehicleTrue(1),x
            canvas.DoMapGraphics(robot) # Draw estimated poitns (in black) and ellipses
            plt.axis([-MapSize-5, MapSize+5, -MapSize-5, MapSize+5]) # Set limits again
            #plt.draw()
#           plt.savefig(str(k)+'.jpg')
            clear_output(wait=True)
            display(canvas.fig)

            if mode == 'step_by_step':
                plt.waitforbuttonpress(-1)
            elif mode == 'visualize_process':
                time.sleep(0.2)
            elif mode == 'non_stop':
                pass # non stop!

    # Final drawings
    %matplotlib inline
    if logger is not None:
        logger.plot()

    # Print map evolution each 5 steps
    for i in range(0,len(log_xEst)):
        with np.printoptions(precision=3):
            print('Iteration: ' + str(i*5))
            print('Estimated xEst:\n' + str(log_xEst[i]))
            print('Estimated PEst:\n' + str(log_PEst[i]))
            print('---------------------------------')
```

In [ ]:

```
#mode = 'step_by_step'
mode = 'visualize_process'
#mode = 'non_stop'

# WORLD MAP
# Num features/landmarks considered within the map
nLandmarks = 1
# Generation of the map
MapSize = 100
Map = MapSize*random.rand(2,nLandmarks)-MapSize/2

# ROBOT
# Covariances for our very bad&expensive sensor (in the system <d,theta>)
Sigma_r = 8.0
Sigma_theta = 8*np.pi/180
```

```
# Initial robot pose
xVehicleTrue = np.vstack([-MapSize/1.5, -MapSize/1.5, 0.]) # We know the exact robot pose

robot = EFKMappingRobot(xVehicleTrue, Sigma_r, Sigma_theta, nLandmarks)

demo_ekf_mapping(robot, Map ,nLandmarks, mode=mode)
```

Considering a larger number of landmarks

Once our EKF implementation is working with one landmark, let's try it in a scenario with 5 landmarks. Again, the content of the `xEst` and `Pest` is shown after each 5 iterations of the algorithm.

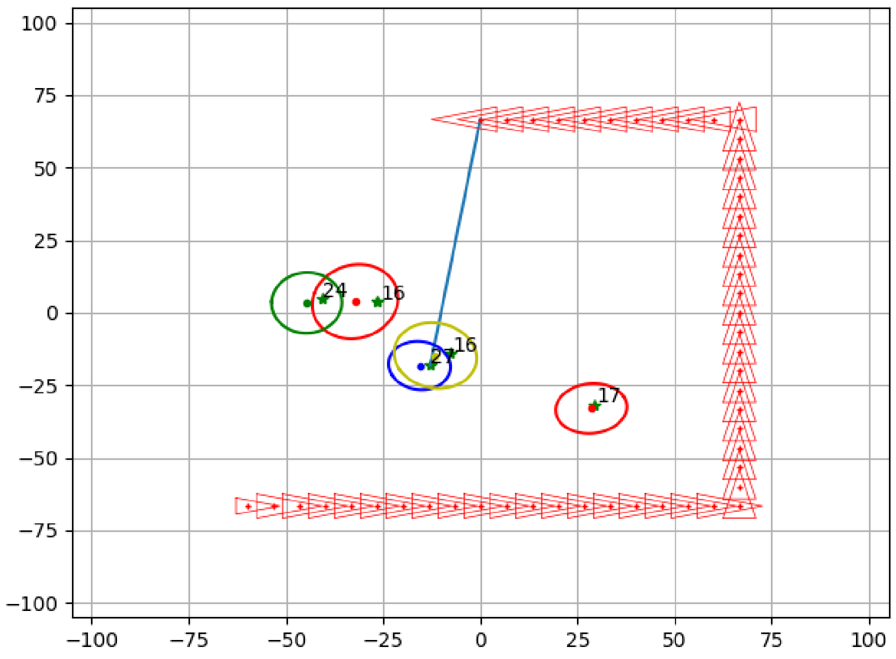

Fig. 2: Execution of the EKF algorithmn for mapping (multiple landmarks).
Same as in Fig 1., each landmark is accompanied by a number of times observed.

In [ ]:
```
#mode = 'step_by_step'
mode = 'visualize_process'
#mode = 'non_stop'

# WORLD MAP
# Num features/landmarks considered within the map
nLandmarks = 5
# Generation of the map
MapSize = 100
Map = MapSize*random.rand(2,nLandmarks)-MapSize/2

# ROBOT
# Covariances for our very bad&expensive sensor (in the system <d,theta>)
Sigma_r = 8.0
Sigma_theta = 7*np.pi/180
# Initial robot pose
xVehicleTrue = np.vstack([-MapSize/1.5, -MapSize/1.5, 0.]) # We know the exact robot pose

robot = EFKMappingRobot(xVehicleTrue, Sigma_r, Sigma_theta, nLandmarks)

demo_ekf_mapping(robot, Map ,nLandmarks, mode=mode)
```

Thinking about it (1)

Having completed these trials, you will be able to **answer the following questions**:

In the **one landmark** case:

- Discuss the evolution of the variables `xEst` and `Pest`, including their dimensions.

  como solo tenemos 1 landmark la primera dimensión de lavariable xEst es (1,2) y su matriz de covarianza (2,2), con más observaciones tendería a ser mejor la estimación y su incertidumbre decrecería

In the **five landmarks** case:

- Why and how the content of the variables `xEst` and `Pest` has change? Discuss their size.

  *Sus dimensiones son: (2*Número de los landmarks observados, 1) y (2xnum de los landmarks observados, 2xnum de los landmarks observados), en cada iteracion la estimacion tiende a mejorar y su incertidumbre decrece*

- What structure does the matrix of covariances have? Is there any kind of correlation among the observations of different landmarks?

  tiene una estructura de submatrices 2x2 en su diagonal asociadas a un landmark cada una, los demas valores son 0 lo que implica que no hay relación entre las observaciones de los diferentes landmarks

Getting performance results

As is normal, the contracting company requires some information about how well our EFK implementation performs. For that, your colleagues have implemented a logger, which is meant to store some information each loop regarding the method performance and plot it at the end of its execution. **Execute the following code cells and take a look at that plots!**

In [ ]:
```python
import pandas as pd

class Logger():
    """ Logs info about the covariance and error of a map.

        Attrs:
            n_features: Number of features in the world.
            log_error: Matrix to store the error in the fitting for each landmark.
            log_det: Matrix to store the determinant of the covariance matrix for each la

    """
    def __init__(self, n_steps, n_features):
        """ Initializes each matrix to log the information

            Args:
                n_steps: Maximum number of steps our robot will take.
                n_features: Number of features in the world.
        """
        self.n_features = n_features
        self.log_error = np.empty((n_steps,n_features))
        self.log_det = np.empty((n_steps,n_features))

    def log(self, k: int, robot: EFKMappingRobot, Map: np.ndarray):
        """ Computes relevant info about the error and covariances.

            It is called once per loop in the demo.
            Args:
                k: Number of iteration we are at. Range: [0, n_steps)
                robot:
                Map:
        """
        for idx in range(self.n_features):
            tid = robot.MappedLandmarks[idx,0]
            if tid <= -1:
                self.log_error[k,idx]= np.Inf
                self.log_det[k,idx]=np.Inf
            else:
                self.log_det[k,idx] = np.linalg.det(robot.PEst[tid:tid+2,tid:tid+2])
                self.log_error[k,idx] = np.sqrt(np.sum((robot.xEst[tid:tid+2,0] - Map[:,

    def plot(self):
        """ Plot all relevant figures. It is called at the end of the demo"""
        fig1 , ax1 =plt.subplots(1, 1, sharex=True)
        fig2 , ax2 =plt.subplots(1, 1, sharex=True)
        fig2.tight_layout()
        fig1.tight_layout()

        df1 = pd.DataFrame(data= self.log_error, columns = ['Landmark {}'.format(i) for
        ax1.set_title('Error between map and est')
        df1.plot(ax = ax1)
        df2 = pd.DataFrame(data=np.log(self.log_det), columns=['Landmark {}'.format(i) f
        ax2.set_title('Det. of covar.')
        df2.plot(ax = ax2)
```

In [ ]:
```
#mode = 'step_by_step'
mode = 'visualize_process'
#mode = 'non_stop'

# WORLD MAP
# Num features/landmarks considered within the map
nLandmarks = 5
# Generation of the map
MapSize = 100
Map = MapSize*random.rand(2,nLandmarks)-MapSize/2

# ROBOT
# Covariances for our very bad&expensive sensor (in the system <d,theta>)
Sigma_r = 8.0
Sigma_theta = 7*np.pi/180
# Initial robot pose
xVehicleTrue = np.vstack([-MapSize/1.5, -MapSize/1.5, 0.]) # We know the exact robot pos

robot = EFKMappingRobot(xVehicleTrue, Sigma_r, Sigma_theta, nLandmarks)

nSteps=100
logger = Logger(n_features=nLandmarks, n_steps=nSteps)

demo_ekf_mapping(robot,
                 Map,
                 nLandmarks,
                 logger=logger,
                 mode='non_stop',
                 nSteps=nSteps)
```

Thinking about it (2)

Having taken a look at the logger and its output, you will be able to **answer the following questions**:

- What information is shown in the figures produced by the logger?

  1º-> el error entre las posiciones estimadas y reales de cada landmark. 2º->El determinante de cada submatriz de cada landmark

- The information about the error and the determinant of the covariance is provided for first time at different iterations of the algorithm for each landmark. Is that an error? Why is this happening?

  No. Ya que antes de esa primera observacion no había información

- The error associated to each landmark not always decreases with new observations. Why could this happen?

  por el ruido de las observaciones del sensor

- On the contrary, the determinant of the covariance matrix associated to each landmark always decreases when new observations are available. Is this an error? Why?

  no, al combinarlos en este caso la incertidumbre decrece