

5.1 Least Squares Global Localization

Least Squares Positioning is a well-known algorithm for estimating the robot localization x given a set of known landmarks in a map. Least Squares is akin to find the best pose \hat{x} by solving a system of equations of the form:

$$z_{m \times 1} = H_{m \times n} \cdot x_{n \times 1}$$

where:

- n is the length of the pose ($n = 3$ in our case, position plus orientation),
- m represents the number of observations, and
- H is the matrix that codifies the observation model relating the measurement z with the robot pose x .

This simple concept, nevertheless, has to be modified in order to be used in real scenarios:

5.1.1 Pseudo-inverse

Generally, to solve an equation system, we only need as many equations as variables. In the robot localization problem, each observation z sets an equation, while the variables are the components of the state/pose, x .

In such a case, where $n = m$, a direct attempt to this problem exists:

$$x = H^{-1} z$$

So a unique solution exists if H is invertible, that is, H is a square matrix with $\det(H) \neq 0$.

However, in real scenarios typically there are available more observations than variables. An approach to address this could be to drop some of the additional equations, but given that observations z are inaccurate (they have been affected by some noise), we may use the additional information to try to mitigate such noise. However, by doing that H is no a squared matrix anymore, hence not being invertible.

Two tools can help us at this point. The first one is the utilization of **Least Squares** to find the closest possible \hat{x} , i.e. the one where the error ($e = Hx - z$) is minimal:

$$\hat{x} = \arg \min_x e^T e = [(z - Hx)^T (z - Hx)] = \arg \min_x ||z - Hx||^2$$

which has a close form solution using the **pseudo-inverse** of a matrix:

$$\hat{x} = \underbrace{(H^T H)^{-1} H^T}_{\text{pseudo-inverse } (H^+) } z$$

The **pseudo-inverse**, in contrast to the normal inverse operation, can be used in non-square matrices!

```
In [1]: %%matplotlib widget
        %%matplotlib inline

        # IMPORTS

        import math

        import numpy as np
        from numpy import linalg
        import matplotlib
        matplotlib.use('TkAgg')
        import matplotlib.pyplot as plt
        import scipy
        from scipy import stats

        import sys
        sys.path.append("..")
        from utils.PlotEllipse import PlotEllipse
        from utils.DrawRobot import DrawRobot
        from utils.tcomp import tcomp
        from utils.tinv import tinv, jac_tinv1 as jac_tinv
        from utils.Jacobians import J1, J2
```

ASSIGNMENT 1: Playing with a robot in a corridor

The following code illustrates a simple scenario where a robot is in a corridor looking at a door, which is placed at the origin of the reference system (see Fig.1). The robot is equipped with a laser scanner able to measure distances, and takes a number of observations z . The robot is placed 3 meters away from the door, but this information is unknown for it. **Your goal is** to estimate the position of the robot in this 1D world using such measurements.



Fig. 1: Simple 1D scenario with a robot equipped with a laser scanner measuring distance to a door.

The following code cell shows the dimensions of all the actors involved in LS-positioning. Complete it for computing the robot pose x from the available information. Recall [np.linalg.inv\(\)](https://numpy.org/doc/stable/reference/generated/numpy.linalg.inv.html) (<https://numpy.org/doc/stable/reference/generated/numpy.linalg.inv.html>).

Note that with this configuration the laser is directly measuring the robot position. If the laser was measuring distances an object placed at a different location m_i , then the equation should be $z_i = Hx - m_i$.

```
In [2]: # Set the robot pose to unknown
x = np.vstack(np.array([None]))

# Sensor measurements to the door
z = np.vstack(np.array([3.7, 2.9, 3.6, 2.5, 3.5]))

# Observation model
H = np.ones(np.array([5, 1]))

print ("Dimensions:")
print ("Pose x:          " + str(x.shape))
print ("Observations z:   " + str(z.shape))
print ("Obs. model H:      " + str(H.shape))
print ("H.T@H:             " + str((H.T@H).shape))
print ("inv(H.T@H):        " + str((np.linalg.inv(H.T@H)).shape))
print ("H.T@z :            " + str((H.T@z).shape))

# Do Least Squares Positioning
x = linalg.inv(H.T @ H) @ H.T @ z

print('\nLS-Positioning')
print('x = ' + str(x[0]))
```

```
Dimensions:
Pose x:          (1, 1)
Observations z: (5, 1)
Obs. model H:   (5, 1)
H.T@H:          (1, 1)
inv(H.T@H):     (1, 1)
H.T@z :         (1, 1)
```

```
LS-Positioning
x = [3.24]
```

Expected output

```
x = [3.24]
```

5.1.2 Weighted measurements

In cases where multiple sensors affected by different noise profiles are used, or in those where the robot is using a sensor with a varying error (e.g. typically radial laser scans are more accurate while measuring distances to close objects), it is interesting to weight the contribution of such measurements while retrieving the robot pose. For example, we are going to consider a sensor whose accuracy drops the further the observed landmark is. Given a *covariance* matrix Q describing the error in the measurements, the equations above are rewritten as:

$$\hat{x} = \arg \min_x e^T Q^{-1} e = [(Hx - z)^T Q^{-1} (Hx - z)]$$

$$\hat{x} \leftarrow (H^T Q^{-1} H)^{-1} H^T Q^{-1} z \quad (1. \text{ Best estimation})$$

$$\Sigma_{\hat{x}} \leftarrow (H^T Q^{-1} H)^{-1} \quad (2. \text{ Uncertainty of the estimation})$$

Example with three measurements having different uncertainty ($\sigma_1^2, \sigma_2^2, \sigma_3^2$):

$$e^T Q^{-1} e = [e_1 \ e_2 \ e_3] \begin{bmatrix} 1/\sigma_1^2 & 0 & 0 \\ 0 & 1/\sigma_2^2 & 0 \\ 0 & 0 & 1/\sigma_3^2 \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix} = \frac{e_1^2}{\sigma_1^2} + \frac{e_2^2}{\sigma_2^2} + \frac{e_3^2}{\sigma_3^2} = \sum_{i=1}^m \frac{e_i^2}{\sigma_i^2}$$

ASSIGNMENT 2: Adding growing uncertainty

We have new information! The manufacturer of the laser scanner mounted on the robot wrote an email telling us that the device is considerably more inaccurate for further distances. Concretely, such uncertainty is characterized by $\sigma^2 = e^z$ (the laser is not so accurate, being polite).

With this new information, implement the computation of the weighted LS-positioning so you can compare the previously estimated position with the new one.

```
In [3]: # Sensor measurements to the door
z = np.vstack(np.array([3.7, 2.9, 3.6, 2.5, 3.5]))

# Uncertainty of the measurements
Q = np.eye(5)*np.exp(z)

# Observation model
H = np.ones(np.array([5,1]))

# Do Least Squares Positioning
x = linalg.inv(H.T @ H) @ H.T @ z

# Do Weighted Least Squares Positioning
x_w = linalg.inv(H.T @ linalg.inv(Q) @ H) @ H.T @ linalg.inv(Q) @ z

print('\nLS-Positioning')
print('x = ' + str(x[0]))

print('\nWeighted-LS-Positioning')
print('x = ' + str(np.round(x_w[0],2)))
```

LS-Positioning
x = [3.24]

Weighted-LS-Positioning
x = [3.01]

Expected output

LS-Positioning

 $x = [3.24]$

5.1.3 Non-linear Least Squares

Until now we have assumed that \hat{x} can be solved as a simple system of equations, i.e. H is a matrix. Nevertheless, typically observation models are non-linear, that is: $z = h(x)$, so the problem now becomes:

$$\hat{x} = \arg \min_x ||z - h(x)||^2$$

No close-form solutions exists for this new problem, but we can approximate it iteratively. Considering the Taylor expansion:

$$h(x) = h(x_0 + \delta) = h(x_0) + J_{h_0} \delta$$

Then we can write:

$$||z - h(x)||^2 \cong ||\underbrace{z - h(x_0)}_{\text{error vector } e} - J_{h_0} \delta||^2 = ||e - J_{h_0} \delta||^2$$

$\leftarrow \delta \text{ is unknown, } J_e = -J_{h_0}$

So we can define the equivalent optimization problem:

$$\delta = \arg \min_{\delta} ||e + J_e \delta||^2 \rightarrow \underbrace{\delta}_{n \times 1} = \underbrace{-(J_e^T J_e)^{-1}}_{n \times n} \underbrace{J_e^T}_{n \times m} \underbrace{e}_{m \times 1} \quad (\delta \text{ that makes the previous squared norm minimum})$$

The weighted form of the δ computation results:

$$\delta = (J_e^T Q^{-1} J_e)^{-1} J_e^T Q^{-1} e$$

Where:

- Q is the measurement covariance (*weighted measurement*)
- J_e is the negative of the Jacobian of the observation model at \hat{x} , also known as $\nabla h_{\hat{x}}$
- e is the error of z against $h(\hat{x})$ (computed using the map information).

As commented, there is no closed-form solution for the problem, but we can iteratively approximate it using the **Gauss-Newton algorithm**:

$$\begin{aligned} \hat{x} &\leftarrow (\dots) && (1. \text{ Initial guess}) \\ \delta &\leftarrow (J_e^T Q^{-1} J_e)^{-1} J_e^T Q^{-1} e && (2. \text{ Evaluate delta/increment}) \\ \hat{x} &\leftarrow \hat{x} - \delta && (3. \text{ Update estimation}) \\ \text{if } \delta > \text{tolerance} &\rightarrow \text{goto } (1.) \\ \text{else} &\rightarrow \text{return } \hat{x} && (4. \text{ Exit condition}) \end{aligned}$$

LS positioning in practice

Suppose that a mobile robot equipped with a range sensor aims to localize itself in a map consisting of a number of landmarks by means of Least Squares and Gauss-Newton optimization.

For that, **you are provided with** the class `Robot` that implements the behavior of a robot that thinks that is placed at `pose` (that's its initial guess, obtained by composing odometry commands), but that has a real position `true_pose`. In addition, the variable `cov` models the uncertainty of its movement, and `var_d` represents the variance (noise) of the range measurements. Take a look at it below.

```
In [4]: class Robot(object):
        """ Simulate a robot base and positioning.

        Attrs:
            pose: Position given by odometry (in this case true_pose)
            true_pose: True position, selected by the mouse in this case
            cov: Covariance for the odometry sensor. Used to add noise
            var_d: Covariance (noise) of each range measurement
        """
        def __init__(self,
                      pose: np.ndarray,
                      cov: np.ndarray,
                      desv_d: int = 0):
            # Pose related
            self.true_pose = pose
            self.pose = pose + np.sqrt(cov)@np.random.randn(3, 1)
            self.cov = cov

            # Sensor related
            self.var_d = desv_d**2

        def plot(self, fig, ax, **kwargs):
            DrawRobot(fig, ax, self.pose, color='red', label="Pose estimate")
            DrawRobot(fig, ax, self.true_pose, color="blue", label="Real position")
```

ASSIGNMENT 3a: Computing distances from the robot to the landmarks

Implement the following function to simulate how our robot observes the world. In this case, the landmarks in the map act as beacons: the robot can sense how far away they are without any information about angles. The robot uses a range sensor with the following observation model:

$$z_i = [d_i] = h(m_i, x) = \left[\sqrt{(x_i - x)^2 + (y_i - y)^2} \right] + w_i$$

where m_i stands for the i^{th} landmark, and w_i is a noise added by the sensor.

Consider two scenarios in the function implementation:

- The measurement is carried out with an ideal sensor, so no noise nor uncertainty exists (`cov_d = 0`).

- The measurement comes from a real sensor affected by a given noise ($\text{cov_d} \neq 0$). We are going to consider that the range sensor is more accurate measuring distances to close landmarks than to far away ones. To implement this, consider that the noise grows with the root of the distance to the landmark, so the resultant uncertainty can be retrieved by:

$$\sigma_{\text{dist}} = \sigma \sqrt{z}$$

that is, $\text{np.sqrt}(z) * \text{np.sqrt}(\text{cov_d})$. Recall that the sensor noise is modeled as a gaussian distribution, so you have to define such distribution and take samples from it using the `stats.norm()`.

(<https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.html>) and `rvs()`.

(https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.rv_continuous.rvs.html)

```
In [5]: def distance(pose: np.ndarray, m: np.ndarray, cov_d: int = 0) -> np.ndarray:
        """ Get observations for every landmark in the map.

        In this case our observations are range only.
        If cov_d > 0 then add gaussian noise with var_d covariance

        Args:
            pose: pose (true or noisy) of the robot taking observation
            m: Map containing all landmarks
            cov_d: Covariance of the sensor

        Returns
            z: numpy array containing distances to all obs. It has shape
            """
        z = np.sqrt((m[0]-pose[0])**2 + (m[1]-pose[1])**2) # compute distance

        if cov_d > 0:
            z += scipy.stats.norm.rvs(loc=0, scale=np.sqrt(z)*np.sqrt(cov_d))

        return z
```

Try your brand new function with the following code:

```
In [6]: pose = np.vstack([2, 2, 0.35])
        m = np.array([[ -5, -15], [20, 56], [54, -18]]).T
        cov_d = 0

        # Compute distances from the sensor to the landmarks
        z = distance(pose, m, cov_d)

        # Now consider a noisy sensor
        cov_d = 0.5
        np.random.seed(seed=0)
        z_with_noise = distance(pose, m, cov_d)

        # Show the results
        print('Measurements without noise:' + str(z))
        print('Measurements with noise:    ' + str(z_with_noise))
```

```
Measurements without noise:[18.38477631 56.92099788 55.71355311]
Measurements with noise:   [23.73319805 59.05577186 60.87928514]
```

Expected output

```
Measurements without noise:[18.38477631 56.92099788 55.713553
11]
Measurements with noise:   [23.73319805 59.05577186 60.879285
14]
```

ASSIGNMENT 3b: Implementing the algorithm

Finally, we get to implement the Least Squares algorithm for localization. We ask you to complete the gaps in the following function, which:

- Starts by initializing the Jacobian of the observation function (J_h) and takes as initial guess (x_{Est}) the position at which the robot thinks it is as given by its odometry ($R1.pose$).
- Then, it enters into a loop until convergence is reached, where:
 1. The distances z_{Est} to each landmark from the estimated position x_{Est} are computed. Recall that the map (landmarks positions) are known (w_{map}).
 - The error is computed by subtracting to the observations provided by the sensor z the distances z_{Est} computed at the previous point. Then, the residual error is computed as $e_{residual} = \sqrt{e_x^2 + e_y^2}$.
 - The Jacobian of the observation model is evaluated at the estimated robot pose (x_{Est}). This Jacobian has two columns and as many rows as observations to the landmarks:

$$jH = \begin{bmatrix} \frac{-1}{d_1}(x_1 - x) & \frac{-1}{d_1}(y_1 - y) \\ \frac{-1}{d_2}(x_2 - x) & \frac{-1}{d_2}(y_2 - y) \\ \dots & \dots \\ \frac{-1}{d_n}(x_n - x) & \frac{-1}{d_n}(y_n - y) \end{bmatrix}$$

being $x_{Est} = [x, y]$, $[x_i, y_i]$ the position of the i^{th} landmark in the map, and d the distance previously computed from the robot estimated pose x_{Est} to the landmarks. The jacobian of the error jE is just $-jH$.

- Computes the increment δ ($incr$) and subtract it to the estimated pose (x_{Est}). Note: recall that $\delta = (J_e^T Q^{-1} J_e)^{-1} J_e^T Q^{-1} e$


```

In [7]: def LeastSquaresLocalization(R1: Robot,
                                     w_map: np.ndarray,
                                     z: np.ndarray,
                                     nIterations=10,
                                     tolerance=0.001,
                                     delay=0.5) -> np.ndarray:
    """ Pose estimation using Gauss-Newton for least squares optimization

    Args:
        R1: Robot which pose we must estimate
        w_map: Map of the environment
        z: Observation received from sensor

        nIterations: sets the maximum number of iterations (default 10)
        tolerance: Minimum error difference needed for stopping the algorithm
        delay: Wait time used to visualize the different iterations

    Returns:
        xEst: Estimated pose

    """

    iteration = 0

    # Initialization of useful variables
    incr = np.ones((2, 1)) # Delta
    jH = np.zeros((w_map.shape[1], 2)) # Jacobian of the observation
    xEst = R1.pose #Initial estimation is the odometry position (usualy 0,0)

    # Let's go!
    while linalg.norm(incr) > tolerance and iteration < nIterations:
        #if plotting:
        plt.plot(xEst[0], xEst[1], '+r', markersize=1+math.floor((iteration-1)/5))
        # Compute the predicted observation (from xEst) and their residuals
        zEst = distance(xEst, w_map, 0)

        # 1) TODO: Compute distance to each landmark from xEst (estimate)
        # 2) TODO: error = difference between real observations and predicted
        e = z - zEst
        residual = np.sqrt(e.T@e) #residual error = sqrt(x^2+y^2)

        # 3) TODO: Compute Jacobians with respect (x,y) (slide 13)
        # The jH is evaluated at our current guess (xEst) -> z_p
        jH = ((-1/zEst)*(w_map-xEst[0:2])).T
        jE = -jH

        # The observation variances Q grow with the root of the distance
        Q = np.diag(R1.var_d*np.sqrt(z))

        # 4) TODO: Solve the equation --> compute incr
        incr = linalg.inv(jE.T @ linalg.inv(Q) @ jE) @ jE.T @ linalg.inv(Q)

        plt.plot([xEst[0], xEst[0]+incr[0]], [xEst[1], xEst[1]+incr[1]])
        xEst[0:2] += incr

        print ("Iteration :" + str(iteration))
        print (" delta : " + str(incr))
        print (" residual: " + str(residual))

```

```

iteration += 1

plt.pause(delay)

plt.plot(xEst[0, 0], xEst[1, 0], '*g', markersize=14, label="Final")

return xEst

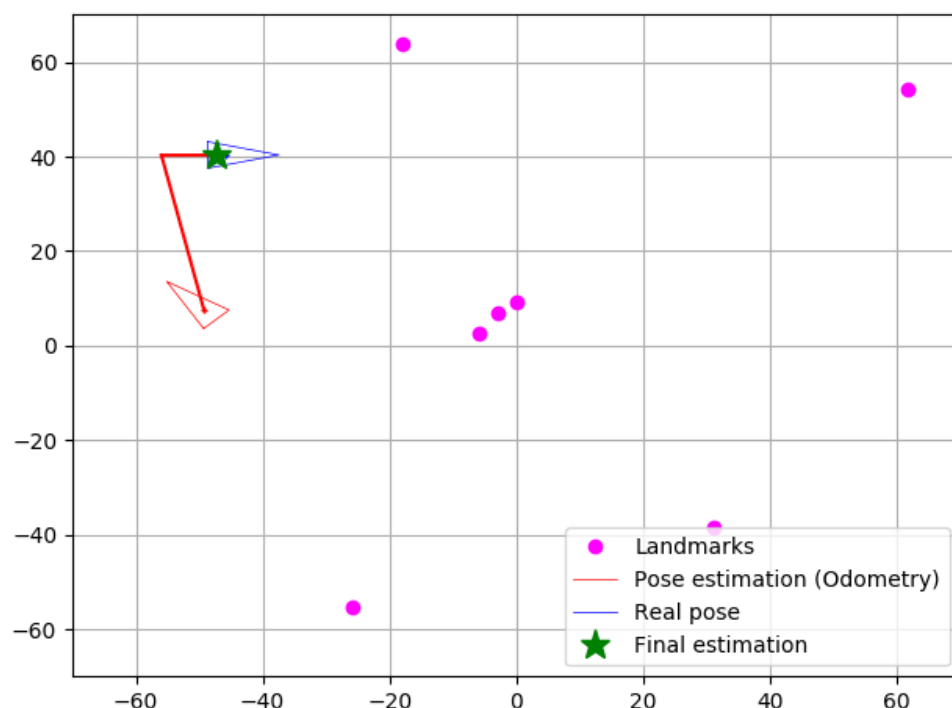
```

The next cell code launches our algorithm, so **we can try it!**. This is done according to the following steps:

1. The map `w_map` is built. In this case, the map consists of a number of landmarks (`nLandmarks`).
2. The program asks the user to set the true position of the robot (`xTrue`) by clicking with the mouse in the map.
3. A new pose is generated from it, `xOdom` , which represents the pose that the robot thinks it is in. This simulates a motion command from an arbitrary pose that ends up with the robot in `xTrue` , but it thinks that it is in `xOdom` .
4. Then the robot takes a (noisy) range measurement to each landmark in the map.
5. Finally, the robot employs a Least Squares definition of the problem and Gauss-Newton to iteratively optimize such a guess (`xOdom`), obtaining a new (and hopefully better) estimation of its pose `xEst` .

Example

The figure below shows an example of execution of this code (once completed).



```

In [15]: def main(nLandmarks=7, env_size=140):
# MATPLOTLIB
fig, ax = plt.subplots()
plt.xlim([-90, 90])
plt.ylim([-90, 90])
plt.grid()
plt.ion()
plt.tight_layout()

fig.canvas.draw()

# VARIABLES
num_landmarks = 7 # number of landmarks in the environment
env_size = 140 # A square environment with x=[-env_size/2, env_size/2]

# MAP CREATION AND VISUALIZATION
w_map = env_size*np.random.rand(2, num_landmarks) - env_size/2 #
ax.plot(w_map[0, :], w_map[1, :], 'o', color='magenta', label="Landmarks")

# ROBOT POSE AND SENSOR INITIALIZATION
desv_d = 0.5 # standard deviation (noise) of the range measurement
cov = np.diag([25, 30, np.pi*180])**2 # covariance of the motion
xStart = np.vstack(plt.ginput(1)).T # get the robot starting point
robot_pose=np.vstack([xStart, 0]) # robot_pose

R1 = Robot(robot_pose, cov, desv_d)
R1.plot(fig, ax)

# MAIN
z = distance(R1.true_pose, w_map, cov_d=R1.var_d) # take (noisy)
LeastSquaresLocalization(R1, w_map, z) # LS Positioning!

# PLOTTING RESULTS
plt.legend()
fig.canvas.draw()

# RUN
main()

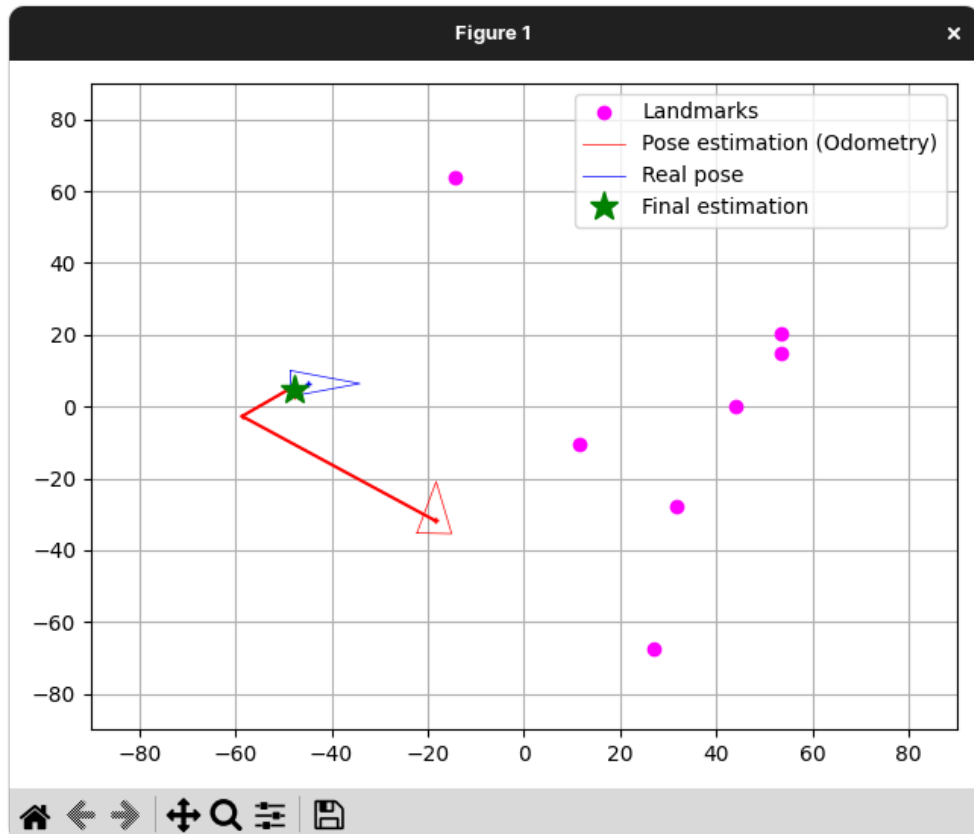
```

```

Iteration :0
  delta :   [20.90463481 10.83892107]
  residual: 46.3412914791478
Iteration :1
  delta :   [ 5.65650599 -2.25683482]
  residual: 11.940471077988429
Iteration :2
  delta :   [ 0.25899406 -0.19405664]
  residual: 6.989144645051092
Iteration :3
  delta :   [ 0.01599978 -0.00132724]
  residual: 6.916431566315218
Iteration :4
  delta :   [ 1.23368423e-03 -8.77552613e-05]
  residual: 6.915625834699756
Iteration :5
  delta :   [ 9.54419700e-05 -6.72070712e-06]
  residual: 6.915570762951273

```

Take a look at the expected results!



Thinking about it (1)

Having completed this notebook above, you will be able to **answer the following questions**:

- What are the dimensions of the error residuals? Does they depend on the number of observations?
El error residual es un valor, por lo que tiene dimensión 1x1. Tampoco podemos decir que dependa del número de observaciones.
- Why is Weighted LS obtaining better results than LS?
Gracias a la ponderación de las observaciones, podemos "dar o quitar" relevancia a aquellas que consideremos más o menos fiables, respectivamente. Esta idea funciona porque en realidad los sensores tienen asociados sus propios valores de incertidumbre, haciendo que se cometan diferentes errores entre sus mediciones.
- Which is the minimum number of landmarks needed for localizing the robot? Why?
Para poder localizar el robot, tenemos que haber tomado un total de 3 observaciones (en el caso de medidores de distancias sin ángulo). Esto es debido a que por cada observación trazaremos una circunferencia a su alrededor que representará la posible posición del robot en la que se tomó la observación. Después de haber hecho las tres, nos daremos cuenta que se cortarán en un único punto, que es donde está el robot.
- Play with different "qualities" of the range sensor. Could you find a value for its variance so the LS method fails?

El valor frontera desde donde empieza a fallar el modelo es 1, por lo que valores mayores a este harán erráticas las predicciones. Si queremos que el modelo falle completamente, deberemos proporcionar valores muy altos como 5,6,7... También se puede conseguir con valores menores si los landmarks del mapa están muy agrupados y establecemos la pose real en el centro de ellas (lo he comprobado).

- Play also with different values for the odometry uncertainty. What does this affect?

Afecta a la primera pose inicial estimada (triángulo rojo) de manera que en un principio el robot crea que está en un lugar muy diferente al que se encuentra.

Sin embargo, es capaz de corregir esta gran equivocación con el modelo de mínimos cuadrados (al final acabamos en la pose real, en azul).

5.2 EKF Localization

The Kalman filter is one of the best studied techniques for filtering and prediction of linear systems. Among its virtues, it provides a way to overcome the occasional un-observability problem of the Least Squares approach. Nevertheless, it makes a strong assumption that the two involved process equations (state transition and observation) are linear.

Unfortunately, you should already know that our system of measurements (i.e. the observation function) and motion (i.e. pose composition) are non-linear. Therefore, this notebook focuses from the get-go on the **Extended Kalman Filter**, which is adapted to work with non-linear systems.

The EKF algorithm consists of 2 phases: **prediction** and **correction**.

```
def ExtendedKalmanFilter( $\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$ ) :
```

Prediction.

$$\bar{\mu}_t = g(\mu_{t-1}, u_t) = \mu_{t-1} \oplus u_t \quad (1. \text{ Pose prediction})$$

$$\bar{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + R_t \quad (2. \text{ Uncertainty of prediction})$$

Correction.

$$K_t = \bar{\Sigma}_t H_t^T (H_t \bar{\Sigma}_t H_t^T + Q_t)^{-1} \quad (3. \text{ Kalman gain})$$

$$\mu_t = \bar{\mu}_t + K_t (z_t - h(\bar{\mu}_t)) \quad (4. \text{ Pose estimation})$$

$$\Sigma_t = (I - K_t H_t) \bar{\Sigma}_t \quad (5. \text{ Uncertainty of estimation})$$

```
return  $\mu_t, \Sigma_t$ 
```

Notice that R_t is the covariance of the motion u_t in the coordinate system of the predicted pose (\bar{x}_t), then (Note: J_2 is our popular Jacobian for the motion command, you could also use J_1):

$$R_t = J_2 \Sigma_{u_t} J_2^T \quad \text{with} \quad J_2 = \frac{\partial g(\mu_{t-1}, u_t)}{\partial u_t}$$

Where:

- (μ_t, Σ_t) represents our robots pose, so $x_t \sim N(\mu_t, \Sigma_t)$
- (u_t, Σ_{u_t}) is the motion command received, and its respective uncertainty.
- (z_t, Q_t) are the observations taken, and their covariance.
- G_t and H_t are the Jacobians of the motion model and the observation model respectively:

$$G_t = \frac{\partial g(\mu_{t-1}, u_t)}{\partial x_{t-1}}, \quad H_t = \frac{\partial h(\bar{\mu}_t)}{\partial x_t}$$

In this notebook we are going to play with the EKF localization algorithm using a map of landmarks and a sensor providing range and bearing measurements from the robot pose to such landmarks. Concretely, **we are going to**:

1. Implement a class modeling a **range and bearing sensor** able to take measurements to landmarks.
2. Complete a class that implements the robot behavior after completing **motion commands**.

3. Implement the **Jacobian of the observation model**.
4. With the previous building blocks, implement our own **EKF filter** and see it in action.
5. Finally, we are going to consider a more **realistic sensor** with a given Field of View and a maximum operational range.

```
In [1]: # IMPORTS
import numpy as np
from numpy import random
from numpy import linalg
import matplotlib
matplotlib.use('TkAgg')
from matplotlib import pyplot as plt
from IPython.display import display, clear_output
import time

import sys
sys.path.append("..")
from utils.AngleWrap import AngleWrapList
from utils.PlotEllipse import PlotEllipse
from utils.Drawings import DrawRobot, drawFOV, drawObservations
from utils.Jacobians import J1, J2
from utils.tcomp import tcomp
```

ASSIGNMENT 1: Getting an observation to a random landmark

We are going to implement the `Sensor()` class modelling a range and bearing sensor. Recall that the observation model of this type of sensors is:

$$z_i = \begin{bmatrix} d_i \\ \theta_i \end{bmatrix} = h(m_i, x) = \begin{bmatrix} \sqrt{(x_i - x)^2 + (y_i - y)^2} \\ \text{atan}\left(\frac{y_i - y}{x_i - x}\right) - \theta \end{bmatrix} + w_i$$

where $m_i = [x_i, y_i]$ are the landmark coordinates in the world frame, $x = [x, y, \theta]$ is the robot pose, and the noise w_i follows a Gaussian distribution with zero mean and covariance matrix:

$$\Sigma_S = \begin{bmatrix} \sigma_r^2 & 0 \\ 0 & \sigma_\theta^2 \end{bmatrix}$$

For that, complete the following methods:

- `observe()` : which, given a real robot pose (`from_pose`), returns the measurements to the landmarks in the map (`world`). If `noisy=True`, then a random gaussian noise with zero mean and covariance Σ_S (`cov`) is added to each measurement. *Hint you can use [random.randn\(\)](https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.random.randn.html) (<https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.random.randn.html>) for that.*
- `random_observation()` : that, given again the robot pose (`from_pose`), randomly selects a landmark from the map (`world`) and returns an observation from the range-bearing sensor using the `observe()` method previously implemented. The `noisy` argument is just passed to `observe()`. *Hint: to randomly select a landmark, use [randint\(\)](https://numpy.org/doc/stable/reference/random/generated/numpy.random.randint.html) (<https://numpy.org/doc/stable/reference/random/generated/numpy.random.randint.html>).*


```

In [2]: class Sensor():
        def __init__(self, cov):
            """
            Args:
                cov: covariance of the sensor.
            """
            self.cov = cov

        def observe(self, from_pose, world, noisy=True, flatten=True):
            """Calculate observation relative to from_pose

            Args:
                from_pose: Position(real) of the robot which takes the ob
                world: List of world coordinates of some landmarks
                noisy: Flag, if true then add noise (Exercise 2)

            Returns:
                Numpy array of polar coordinates of landmarks from th
                They are organised in a vertical vector ls = [d_0 , a
                Dims (2*n_landmarks, 1)
            """
            delta = world - from_pose[0:2]

            z = np.empty_like(delta)
            z[0, :] = np.sqrt(delta[0] ** 2 + delta[1] ** 2)
            z[1, :] = np.arctan(delta[1] / delta[0]) - from_pose[2]
            z[1, :] = AngleWrapList(z[1, :])

            if noisy:
                z += np.sqrt(self.cov) @ random.randn(2, z.shape[1])

            if flatten:
                return np.vstack(z.flatten('F'))
            else:
                return z

        def random_observation(self, from_pose, world, noisy=True):
            """ Get an observation from a random landmark

            Args: Same as observe().

            Returns:
                z: Numpy array of obs. in polar coordinates
                landmark: Index of the randomly selected landmark in
                Although it is only one index, you should return
                a numpy array.
            """
            n_landmarks = world.shape[1]
            rand_idx = np.random.randint(0, n_landmarks)
            world = world[:, [rand_idx]]

            z = self.observe(from_pose, world, noisy)

            return z, np.array([rand_idx])

```

You can use the code cell below to test your implementation.

```

In [3]: # TRY IT!
seed = 0
np.random.seed(seed)

# Sensor characterization
SigmaR = 1 # Standard deviation of the range
SigmaB = 0.7 # Standard deviation of the bearing
Q = np.diag([SigmaR**2, SigmaB**2]) # Cov matrix

sensor = Sensor(Q)

# Map
Size = 50.0
NumLandmarks = 3
Map = Size*2*random.rand(2, NumLandmarks)-Size

# Robot true pose
true_pose = np.vstack([-Size+Size/3, -Size+Size/3, np.pi/2])

# Take a random measurement
noisy = False
z = sensor.random_observation(true_pose, Map, noisy)

noisy = True
noisy_z = sensor.random_observation(true_pose, Map, noisy)

# Take observations to every landmark in the map
zs = sensor.observe(true_pose, Map, noisy)

print('Measurement:\n' + str(z))
print('Noisy measurement:\n' + str(noisy_z))
print('Measurements to every landmark in the map:\n' + str(zs))

```

```

Measurement:
(array([[53.76652662],
       [-0.79056712]]), array([0]))
Noisy measurement:
(array([[64.73997127],
       [-0.81342958]]), array([2]))
Measurements to every landmark in the map:
[[ 5.51319938e+01]
 [-1.10770618e+00]
 [ 6.04762304e+01]
 [-1.46219661e+00]
 [ 6.23690518e+01]
 [-5.72010701e-02]]

```

Expected output

Measurement:

```
(array([[53.76652662],
        [-0.79056712]]), array([0]))
```

Noisy measurement:

ASSIGNMENT 2: Simulating the robot motion

In the robot motion chapter we commanded a mobile robot to follow a squared trajectory.

We provide here the `Robot` class that implements:

- how the robot pose evolves after executing a motion command (`step()` method), and
- the functionality needed to graphically show its ideal pose (`pose`), true pose (`true_pose`) and estimated pose (`xEst`) in the `draw()` function.

Your mission is to complete the `step()` method by adding random noise to each motion command (`noisy_u`) based on the following covariance matrix, and update the true robot pose (`true_pose`):

$$\Sigma_{u_i} = \begin{bmatrix} \sigma_{\Delta x}^2 & 0 & 0 \\ 0 & \sigma_{\Delta y}^2 & 0 \\ 0 & 0 & \sigma_{\Delta \theta}^2 \end{bmatrix}$$

Hint: Recall again the `random.randn()` (<https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.random.randn.html>) function.

```
In [4]: class Robot():
        def __init__(self, true_pose, cov_move):
            # Robot description (Starts as perfectly known)
            self.pose = true_pose
            self.true_pose = true_pose
            self.cov_move = cov_move

            # Estimated pose and covariance
            self.xEst = true_pose
            self.PEst = np.zeros((3, 3))

        def step(self, u):
            self.pose = tcomp(self.pose, u) # New pose without noise
            noise = np.sqrt(self.cov_move)@random.randn(3,1) # Generate noise
            noisy_u = u + noise # Apply noise to the control action
            self.true_pose = tcomp(self.true_pose, noisy_u) # New noisy pose

        def draw(self, fig, ax):
            DrawRobot(fig, ax, self.pose, color='r')
            DrawRobot(fig, ax, self.true_pose, color='b')
            DrawRobot(fig, ax, self.xEst, color='g')
            PlotEllipse(fig, ax, self.xEst, self.PEst, 4, color='g')
```

It is time to test your `step()` function!

```
In [5]: # Robot base characterization
SigmaX = 0.8 # Standard deviation in the x axis
SigmaY = 0.8 # Standard deviation in the y axis
SigmaTheta = 0.1 # Bearing standar deviation
R = np.diag([SigmaX**2, SigmaY**2, SigmaTheta**2]) # Cov matrix

# Create the Robot object
true_pose = np.vstack([2,3,np.pi/2])
robot = Robot(true_pose, R)

# Perform a motion command
u = np.vstack([1,2,0])
np.random.seed(0)
robot.step(u)

print('robot.true_pose.T:' + str(robot.true_pose.T) + '\n')

robot.true_pose.T:[[-0.32012577  5.41124188  1.66867013]]'
```

Expected output

```
robot.true_pose.T:[[-0.32012577  5.41124188  1.66867013]]'
```

ASSIGNMENT 3: Jacobians of the observation model

Given that the position of the landmarks in the map is known, we can use this information in a Kalman filter, in our case an EKF. For that we need to implement the **Jacobians of the observation model**, as required by the correction step of the filter.

Implement the function `getObsJac()` that given:

- the predicted pose in the first step of the Kalman filter,
- a number of observed landmarks, and
- the map,

returns such Jacobian. Recall that, for each observation to a landmark:

$$\nabla H = \frac{\partial h}{\partial \{x, y, \theta\}} = \begin{bmatrix} -\frac{x_i - x}{d} & -\frac{y_i - y}{d} & 0 \\ \frac{y_i - y}{d^2} & -\frac{x_i - x}{d^2} & -1 \end{bmatrix}_{2 \times 3}$$

Recall that $[x_i, y_i]$ is the position of the i^{th} landmark in the map, $[x, y]$ is the robot predicted pose, and d the distance from such predicted pose to the landmark. This way, the resultant Jacobian dimensions are $(\#observed_landmarks \times 2, 3)$, that is, the Jacobians are stacked vertically to form the matrix H .

```
In [6]: def getObsJac(xPred, lm, Map):
        """ Obtain the Jacobian for all observations.

        Args:
            xPred: Position of our robot at which Jac is evaluated.
            lm: Numpy array of observations to a number of landmarks
            Map: Map containing the actual positions of the observati

        Returns:
            jH: Jacobian matrix (2*n_landmaks, 3)
        """
        n_land = len(lm)
        jH = np.empty((2*n_land,3))

        for i in range(n_land):
            # Auxiliary variables
            dx = Map[0,lm[i]] - xPred[0,0]
            dy = Map[1,lm[i]] - xPred[1,0]
            d = np.sqrt(dx ** 2 + dy ** 2)
            d2 = d ** 2

            ii = 2*i

            # Build the Jacobian
            jH[ii:ii+2,:] = [
                [-dx/d, -dy/d, 0],
                [dy/d2, -dx/d2, -1]
            ]

        return jH
```

Time to check your function!

```
In [7]: # TRY IT!

observed_landmarks = np.array([0,2])
xPred = np.vstack([-Size+Size/3, -Size+Size/3, np.pi/2]) # Robot pred
jH = getObsJac(xPred, observed_landmarks, Map) # Retrieve the evaluat

print ('Jacobian dimensions: ' + str(jH.shape) )
print ('jH:' + str(jH))
```

```
Jacobian dimensions: (4, 3)
jH:[[-0.71075232 -0.70344235  0.          ]
      [ 0.01308328 -0.01321923 -1.          ]
      [-0.67304061 -0.73960552  0.          ]
      [ 0.01141455 -0.01038723 -1.          ]]
```

Expected output:

```
Jacobian dimensions: (4, 3)
jH:[[-0.71075232 -0.70344235  0.          ]
      [ 0.01308328 -0.01321923 -1.          ]
      [-0.67304061 -0.73960552  0.          ]
      [ 0.01141455 -0.01038723 -1.          ]]
```

ASSIGNMENT 4: Completing the EKF

Congratulations! You now have all the building blocks needed to implement an EKF filter (both prediction and correction steps) for localizing the robot and show the estimated pose and its uncertainty.

For doing that, complete the `EKFLocalization()` function below, which returns:

- the estimated pose (`xEst`), and
- its associated uncertainty (`PEst`),

given:

- the previous estimations (`self.xEst` and `self.PEst` stored in `robot`),
- the features of the sensor (`sensor`),
- the motion command provided to the robot (`u`),
- the observations done (`z`),
- the indices of the observed landmarks (`landmark`), and
- the map of the environment (`Map`).

```

In [8]: def EKFLocalization(robot, sensor, u, z, landmark, Map):
        """ Implement the EKF algorithm for localization

        Args:
            robot: Robot base (contains the state: xEst and PEst)
            sensor: Sensor of our robot.
            u: Motion command
            z: Observations received
            landmark: Indices of landmarks observed in z
            Map: Array with landmark coordinates in the map

        Returns:
            xEst: New estimated pose
            PEst: Covariance of the estimated pose
        """

        # Prediction
        xPred = tcomp(robot.xEst, u)
        G = J1(xPred, robot.xEst)
        j2 = J2(xPred, u)
        PPred = G@robot.PEst@G.T + j2@robot.cov_move@j2.T

        # Correction (You need to compute the gain k and the innovation z)
        if landmark.shape[0] > 0:
            H = getObsJac(xPred, landmark, Map) # Observation Jacobian
            K = PPred @ H.T @ np.linalg.inv((H @ PPred @ H.T) + np.kron(r
            xEst = xPred + K @ (z-sensor.observe(xPred, Map[:,landmark],
            PEst = (np.identity(H.shape[1]) - K @ H) @ PPred # New estima
        else:
            xEst = xPred
            PEst = PPred

        return xEst, PEst

```

You can **validate your code** with the code cell below.

```

In [9]: # TRY IT!

np.random.seed(2)

# Create the map
Map=Size*2*random.rand(2,20)-Size

# Create the Robot object
true_pose = np.vstack([2,3,0])
R = np.diag([0.1**2, 0.1**2, 0.01**2]) # Cov matrix
robot = Robot(true_pose, R)

# Perform a motion command
u = np.vstack([10,0,0])
robot.step(u)

# Get an observation to a landmark
noisy = True
noisy_z, landmark_index = sensor.random_observation(true_pose, Map, r

# Estimate the new robot pose using EKF!
robot.xEst, robot.PEst = EKFLocalization(robot, sensor, u, noisy_z, 1

# Show results!
print('robot.pose.T:' + str(robot.pose.T) + '\n')
print('robot.true_pose.T:' + str(robot.true_pose.T) + '\n')
print('robot.xEst.T:' + str(robot.xEst.T) + '\n')
print('robot.PEst:' + str(robot.PEst.T))

robot.pose.T:[[12.  3.  0.]]'
robot.true_pose.T:[[ 1.20000010e+01  3.05423526e+00 -3.13508197e-0
3]]'
robot.xEst.T:[[ 1.19586407e+01  2.96047951e+00 -1.48514185e-04]]'
robot.PEst:[[ 9.94877200e-03 -4.94253023e-05 -3.18283546e-08]
[-4.94253023e-05  9.95211532e-03  3.29230513e-08]
[-3.18283546e-08  3.29230513e-08  9.99795962e-05]]

```

Expected output:

```

robot.pose.T:[[12.  3.  0.]]'
robot.true_pose.T:[[ 1.20000010e+01  3.05423526e+00 -3.135081
97e-03]]'
robot.xEst.T:[[ 1.19586407e+01  2.96047951e+00 -1.48514185e-0
4]]'
robot.PEst:[[ 9.94877200e-03 -4.94253023e-05 -3.18283546e-08]
[-4.94253023e-05  9.95211532e-03  3.29230513e-08]
[-3.18283546e-08  3.29230513e-08  9.99795962e-05]]

```

Playing with EKF

The following code helps you to see the EKF filter in action!. Press any key on the emerging window to send a motion command to the robot and check how the landmark it observes changes, as well as its ideal, true and estimated poses.

Notice that you can change the value of `seed` within the `main()` function to try different executions.

Example

The figure below shown an example of the execution of the EKF localization algorithm with the code implemented until this point.

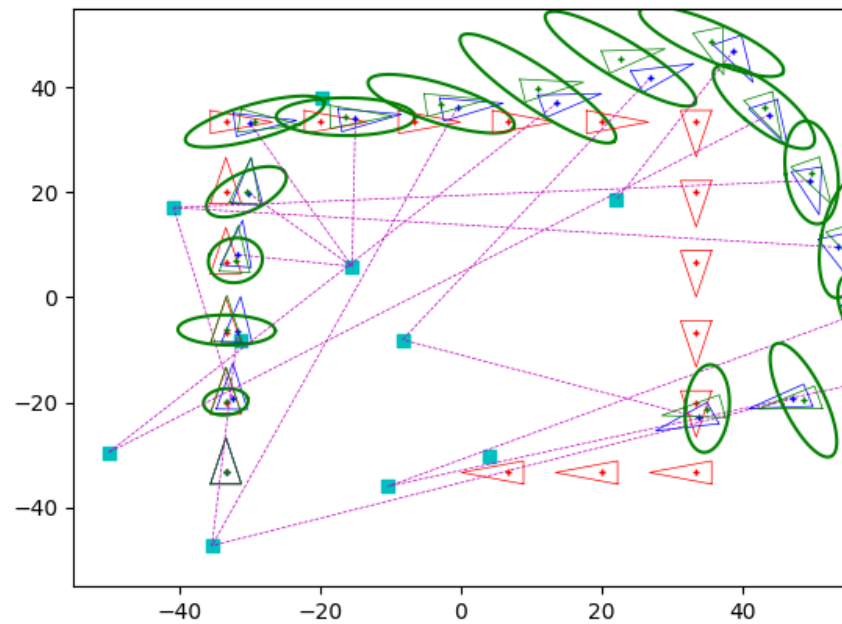


Fig. 1: Execution of the EKF algorithmn for localization, it shows the true (in blue) and expected (in red) poses, along the results from localization: pose and ellipse (in green), the existing landmarks (in cyan), and each observation made (dotted lines).


```

In [10]: def main(robot,
                sensor,
                mode='one_landmark',
                visualization = 'non_stop',
                nSteps=20, # Number of motions
                turning=5, # Number of motions before turning (square path)
                Size=50.0,
                NumLandmarks=10):

    seed = 1
    np.random.seed(seed)

    #Create map
    Map=Size*2*random.rand(2,NumLandmarks)-Size

    # MATPLOTLIB
    if visualization == 'non_stop':
        %matplotlib widget
    elif visualization == 'step_by_step':
        #%matplotlib inline
        matplotlib.use('TkAgg')
        plt.ion()

    fig, ax = plt.subplots()
    plt.plot(Map[0,:],Map[1,:], 'sc')
    plt.axis([-Size-15, Size+15, -Size-15, Size+15])
    plt.title(mode)

    robot.draw(fig, ax)
    fig.canvas.draw()

    # MAIN LOOP

    u = np.vstack([(2*Size-2*Size/3)/turning,0,0]) # control action

    if visualization == 'step_by_step':
        plt.waitforbuttonpress(-1)

    for k in range(0, nSteps-3): # Main loop
        u[2] = 0
        if k % turning == turning-1: # Turn?
            u[2] = -np.pi/2

        robot.step(u)

        # Get sensor observation/s
        if mode == 'one_landmark':
            # DONE (Exercise 4)
            z, landmark = sensor.random_observation(robot.true_pose,
            ax.plot(
                [robot.true_pose[0,0], Map[0,landmark]],
                [robot.true_pose[1,0], Map[1,landmark]],
                color='m', linestyle="--", linewidth=.5)
        elif mode == 'landmarks_in_fov':
            # DONE (Exercise 5)
            z, landmark = sensor.observe_in_fov(robot.true_pose, Map)
            drawObservations(fig, ax, robot.true_pose, Map[:, landmark])

        robot.xEst, robot.PEst = EKFLocalization(robot, sensor, u, z,

    # Drawings

```

```
# Plot the FOV of the robot
if mode == 'landmarks_in_fov':
    h = sensor.draw(fig, ax, robot.true_pose)
#end

robot.draw(fig, ax)
fig.canvas.draw()

if visualization == 'non_stop':
    clear_output(wait=True)
    display(fig)
elif visualization == 'step_by_step':
    plt.waitforbuttonpress(-1)

if mode == 'landmarks_in_fov':
    h.pop(0).remove()
    fig.canvas.draw()

if visualization == 'non_stop':
    plt.close()
elif visualization == 'step_by_step':
    plt.ioff()
```

```

In [11]: # RUN
mode = 'one_landmark'
# mode = 'landmarks_in_fov'
visualization = 'non_stop'
# visualization = 'step_by_step'

Size=50.0

# Robot base characterization
SigmaX = 0.8 # Standard deviation in the x axis
SigmaY = 0.8 # Standard deviation in the y axis
SigmaTheta = 0.1 # Bearing standar deviation
R = np.diag([SigmaX**2, SigmaY**2, SigmaTheta**2]) # Cov matrix

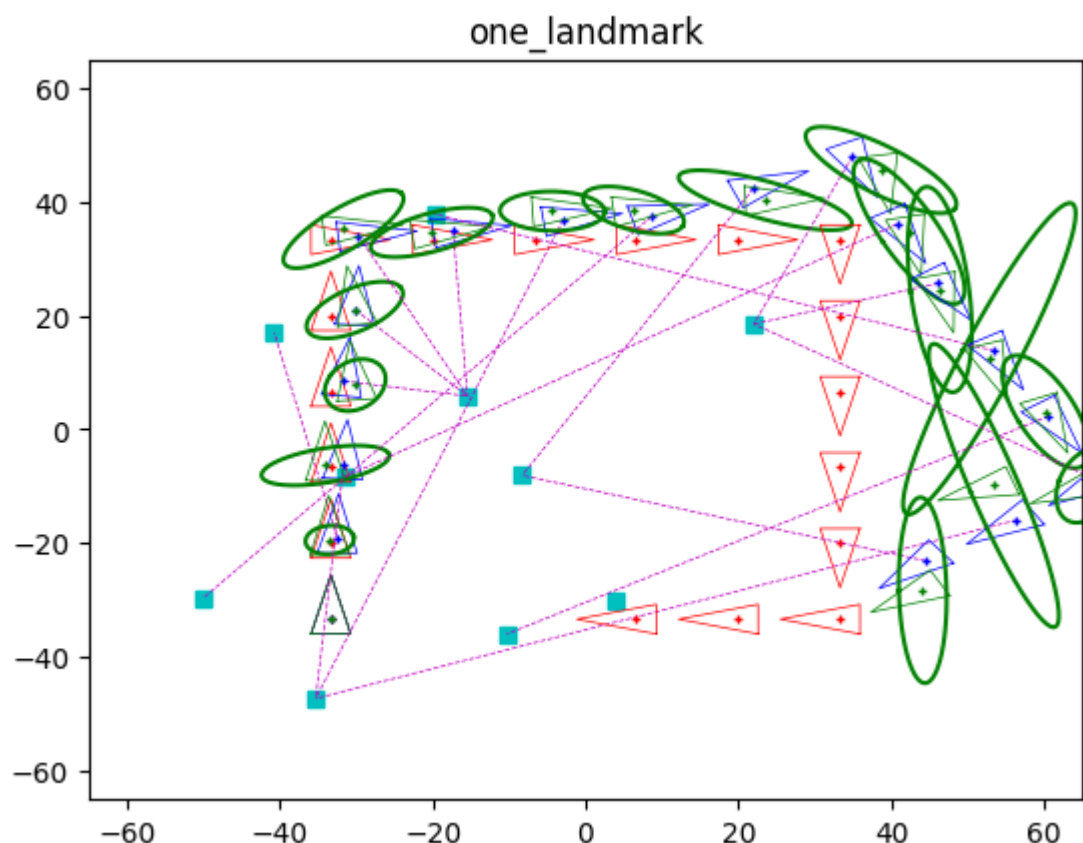
true_pose = np.vstack([-Size+Size/3, -Size+Size/3, np.pi/2])
robot = Robot(true_pose, R)

# Sensor characterization
SigmaR = 1 # Standard deviation of the range
SigmaB = 0.7 # Standard deviation of the bearing
Q = np.diag([SigmaR**2, SigmaB**2]) # Cov matrix

sensor = Sensor(Q)

main(robot, sensor, mode=mode, visualization=visualization, Size=Size

```



ASSIGNMENT 5: Implementing the FoV of a sensor.

Sensors exhibit certain physical limitations regarding their field of view (FoV) and maximum operating distance (max. Range). Besides, these devices often do not deliver measurements from just one landmark each time, but from all those landmarks in the FoV.

The `FoVSensor()` class below extends the `Sensor()` one to implement this behaviour. Complete the `observe_in_fov()` method to consider that the sensor can only provide information from the landmarks in a limited range r_l and a limited orientation $\pm\alpha$ with respect to the robot pose. For that:

1. Get the observations to every landmark in the map. Use the `observe()` function previously implemented for that, but with the argument `flatten=False`. With that option the function returns the measurements as:

$$z = \begin{bmatrix} d_1 & \cdots & d_m \\ \theta_1 & \cdots & \theta_m \end{bmatrix}$$

2. Check which observations lay in the sensor FoV and maximum operating distance.
Hint: for that, you can use the `np.asarray()` (<https://docs.scipy.org/doc/numpy/reference/generated/numpy.asarray.html>) function with the conditions to be fulfilled by the valid measurements inside, and then filter the results with `np.nonzero()` (<https://docs.scipy.org/doc/numpy/reference/generated/numpy.nonzero.html>).
3. Flatten the resultant matrix `z` to be again a vector, so it has the shape $(2 \times \text{\#Observed_landmarks}, 1)$. *Hint: take a look at `np.ndarray.flatten()` (<https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.flatten.html>) and choose the proper argument.*

Notice that it could happen that any landmark exists in the field of view of the sensor, so the robot couldn't gather sensory information in that iteration. This, which is a problem using Least Squares Positioning, is not an issue with EKF. **Hint: you can change the value of `seed` within the `main()` function to try different executions**

```

In [12]: class FOVSensor(Sensor):
    def __init__(self, cov, fov, max_range):
        super().__init__(cov)
        self.fov = fov
        self.max_range = max_range

    def observe_in_fov(self, from_pose, world, noisy=True):
        """ Get all observations in the fov

        Args:
            from_pose: Position(real) of the robot which takes the ob
            world: List of world coordinates of some landmarks
            noisy: Flag, if true then add noise (Exercise 2)

        Returns:
            Numpy array of polar coordinates of landmarks from the pe
            They are organised in a vertical vector ls = [d_0 , a_0,
            Dims (2*n_landmarks, 1)
        """
        # 1. Get observations to every landmark in the map WITHOUT NO
        z = self.observe(from_pose, world, False, False)

        # 2. Check which ones lay on the sensor FOV
        angle_limit = self.fov/2 # auxiliar variable
        feats_idx = np.nonzero((z[1]<angle_limit) & (z[1]>-angle_lim

        if noisy:
            # 1. Get observations to every landmark in the map WITH N
            z = self.observe(from_pose, world, True, False)

        z = z[:, feats_idx] # extracts the valid observations from z

        # 3. Flatten the resultant vector of measurements so z=[d_1,
        if z.size>0:
            z = np.vstack(z.flatten('F'))

        return z, feats_idx

    def draw(self, fig, ax, from_pose):
        """ Draws the Field of View of the sensor from the robot pose
        return drawFOV(fig, ax, from_pose, self.fov, self.max_range)

```

You can now **try** your new and more realistic sensor.

```

In [15]: # TRY IT!
np.random.seed(0)

# Create the sensor object
cov = np.diag([0.1**2, 0.1**2]) # Cov matrix
fov = np.pi/2
max_range = 2
sensor = FOVSensor(cov, fov, max_range)

# Create a map with three landmarks
Map = np.array([[2., 2.5, 3.5, 0.5],[2., 3., 1.5, 3.5]])

# Take an observation of landmarks in FoV
robot_pose = np.vstack([1.,2.,0.])
z, feats_idx = sensor.observe_in_fov(robot_pose, Map)
print('z:' +str(z))

# Plot results
fig, ax = plt.subplots()
plt.axis([0, 5, 0, 5])
plt.title('Measurements to landmarks in sensor FOV')
plt.plot(Map[0,:],Map[1,:],'sc')
sensor.draw(fig, ax, robot_pose)
drawObservations(fig, ax, robot_pose, Map[:, feats_idx])
DrawRobot(fig,ax,robot_pose)

z:[[1.17640523]
 [0.1867558 ]
 [1.84279136]
 [0.49027482]]

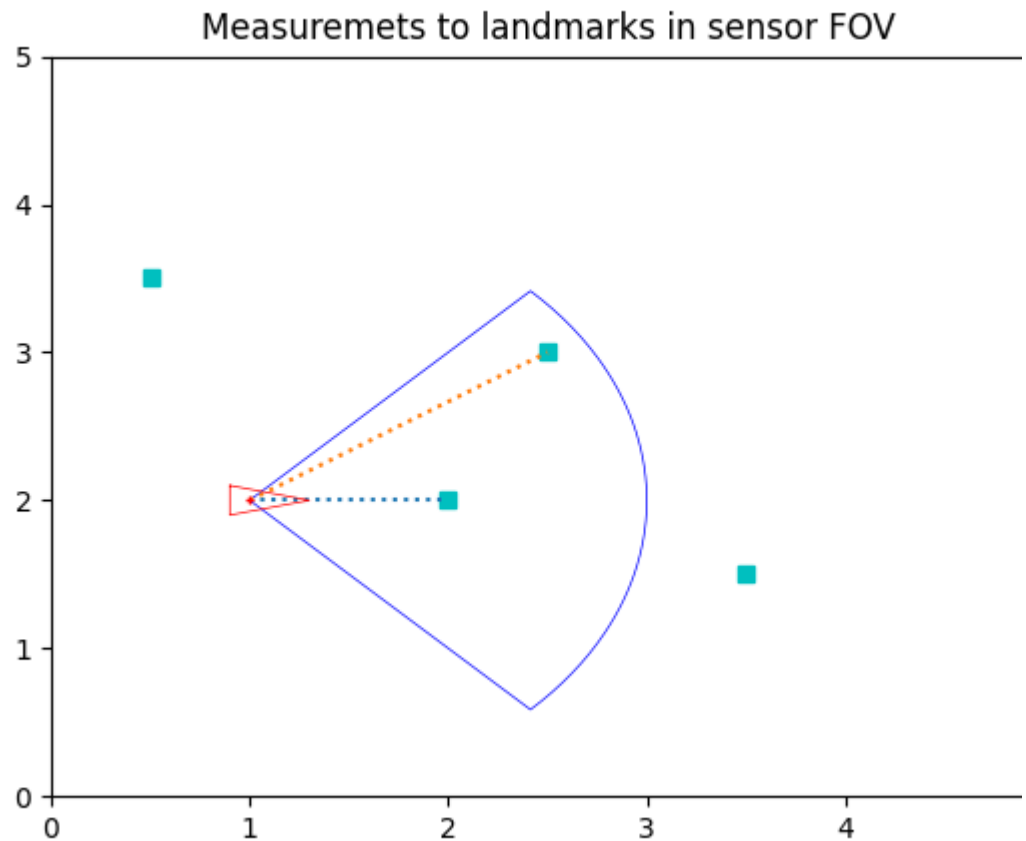
```

```

Out[15]: [<matplotlib.lines.Line2D at 0x7f69ae32a0d0>]

```


Figure 1



Expected output:

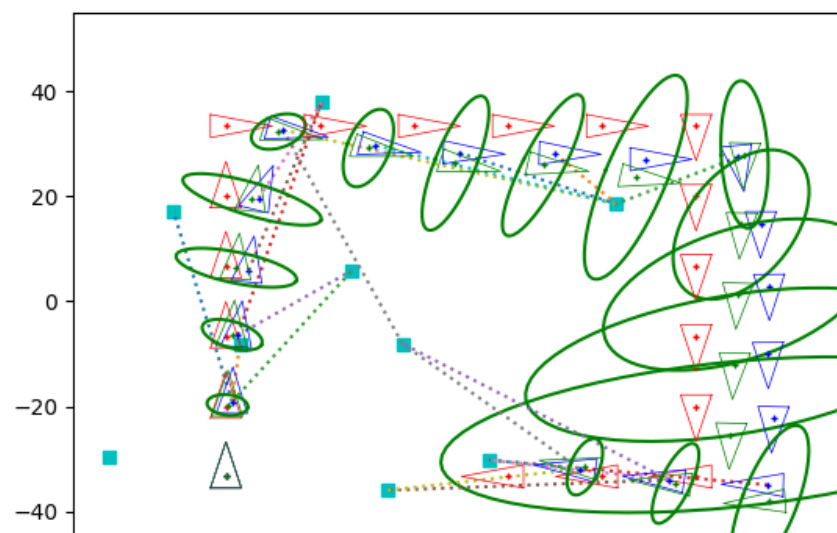
```
z: [[1.17640523]
     [0.1867558 ]
     [1.84279136]
     [0.49027482]]
```

Playing with EKF and the new sensor

And finally, play with your own FULL implementation of the EKF filter with a more realistic sensor :)

Example

The figure below shows an example of the execution of EKF using information from all the landmarks within the FOV:



```

In [17]: # RUN
#mode = 'one_landmark'
mode = 'landmarks_in_fov'
visualization = 'non_stop'
#visualization = 'step_by_step'
Size=50.0

# Robot base characterization
SigmaX = 0.8 # Standard deviation in the x axis
SigmaY = 0.8 # Standard deviation in the y axis
SigmaTheta = 0.1 # Bearing standar deviation
R = np.diag([SigmaX**2, SigmaY**2, SigmaTheta**2]) # Cov matrix

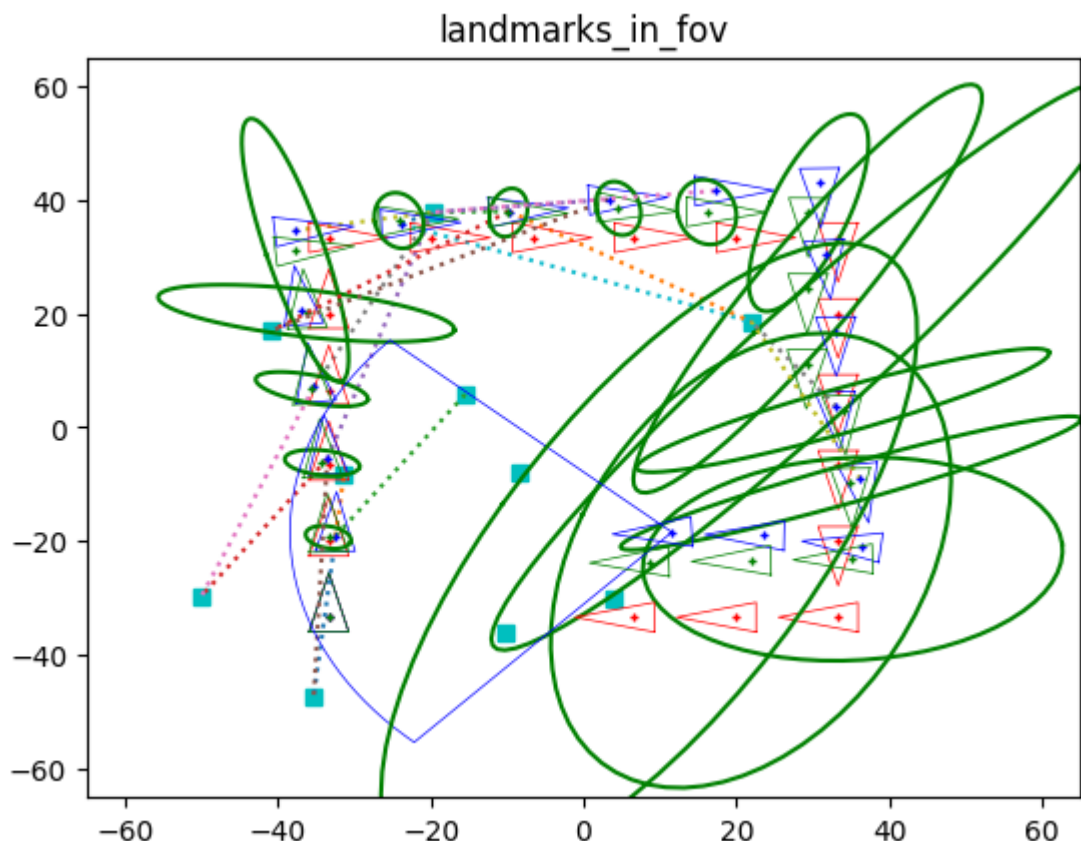
true_pose = np.vstack([-Size+Size/3, -Size+Size/3, np.pi/2])
robot = Robot(true_pose, R)

# Sensor characterization
SigmaR = 1 # Standard deviation of the range
SigmaB = 0.7 # Standard deviation of the bearing
Q = np.diag([SigmaR**2, SigmaB**2]) # Cov matrix
fov = np.pi/2 # field of view = 2*alpha
max_range = Size # maximum sensor measurement range

sensor = FOVSensor(Q, fov, max_range)

main(robot, sensor, mode=mode, visualization=visualization, Size=Size)

```



Thinking about it (1)

Having completed the EKF implementation, you are ready to **answer the following questions**:

- What are the dimensions of the Jacobians of the observation model (matrix H)? Why?

*Su dimensión es $[\text{num_landmarks} * 2, 3]$ puesto que es el resultado de stackear verticalmente los Jacobianos de las observaciones para formar la matriz H .*

- Discuss the evolution of the ideal, true and estimated poses when executing the EKF filter (with the initial sensor).

Mientras que la pose ideal (rojo) es justamente lo que tendr a que ocurrir si no hubiera ruido, la pose verdadera (azul) s  est  afectada por esa perturbaci n, por lo que se producen desviaciones en la trayectoria del robot. La pose estimada (verde) es el c culo que EKF hace en funci n de las observaciones a los landmarks visibles por el robot durante su trayecto. A medida que el robot los observa, la incertidumbre con respecto a su posici n disminuye. Si no ve (casi) ninguno, literalmente se pierde y crece en gran medida.

- Discuss the evolution of the ideal, true and estimated poses when executing the EKF filter (with the sensor implementing a FOV). Pay special attention to their associated uncertainties.

Lo novedoso de este sensor es que se implementa un rango de visi n que limita las observaciones a los landmarks que nuestro robot puede percibir. Como consecuencia de esto, las correcciones se producir n solo si el landmark se encuentra en el rango del sensor (lo cual es bastante realista). Un menor numero de correcciones hace que la incertidumbre crezca bastante m s hasta que no veamos un landmark, que har  que decrezca.

- What happens in the EKF filter when the robot performs a motion command, but it is unable to measure distances to any landmark, i.e. they are out of the sensor FOV?

Aunque la incertidumbre es enorme (nuestro robot va a ciegas), el filtro EKF sigue siendo capaz de estimar la nueva pose con un sorprendente acercamiento a la pose real.