3.3. Odometry-based motion model

I completed the optional tasks of this notebook. I just did what it was asked to do. I did not add anything else

Odometry can be defined as the sum of wheel encoder pulses (see Fig. 1) to compute the robot pose. In this way, most robot bases/platforms provide some form of *odometry information*, a measurement of how much the robot has moved in reality. It is fun to know that cdometry comes from the Greek words ὁδός [odos] (route) and μέτρον [metron] (measurement), which mean *measurement of the route*.

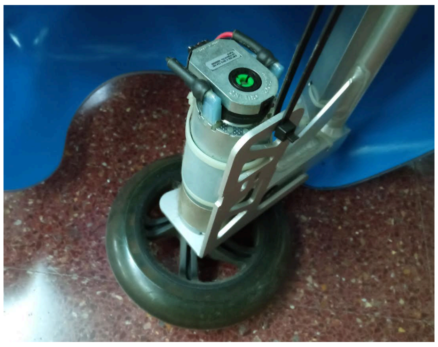


Fig. 1: Example of a wheel encoder used to sum pulses and compute the robot pose.

Such information is yielded by the firmware of the robotic base, which computes it at very high rate (*e.g.* at 100Hz) considering constant linear v_t and angular w_t velocities. Concretely, if we know the total number of markers n_{total} (empty holes in the mask) the encoder has, the angle that the wheel turns per marker can be computed as:

$$\alpha = \frac{2\pi}{n_{total}} \text{ (radians)}$$

This angle increment is detected each time a pulse occurs. Then, in a given time interval Δt, the total angle rotated by the wheel given the number of pulses detected n_t is:

$$\Delta\beta_t = n_t \cdot \alpha \text{ (radians)}$$

This way, the angular velocity ω of the wheel can be computed as:

$$\omega \simeq \frac{\Delta\beta_t}{\Delta t} \text{ (radians/seconds)}$$

Note that this angular is speed is different from the one w.r.t. the ICR. Since we are considering a differential drive locomotion system, the pose increment can be retrieved as:

$$\Delta p = \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta \theta \end{bmatrix} = \begin{bmatrix} \frac{v_p}{w} sin(w\Delta t) \\ \frac{v_p}{w}[1 - cos(w\Delta t)] \\ w\Delta t \end{bmatrix}$$

being $w = \frac{v_r - v_l}{l}$ the angular velocity of the robot w.r.t. the ICR (with l the distance between the wheels), v_r and v_l the linear velocities of the right and left wheels respectly, that can be computed from the previously obtained angular velocities ω_r and ω_l with $v = r \cdot \omega$ (r stands for the wheel radius), and v_p the linear velocity at the robot-axis midpoint that can be computed as $v_p = \frac{v_l + v_r}{2}$.

As commented, the firmware of the robotic base computes these pose increments at a very high rate, and makes it available to the robot at lower rate (*e.g.* 10Hz) using a tool that we already know: the composition of poses:

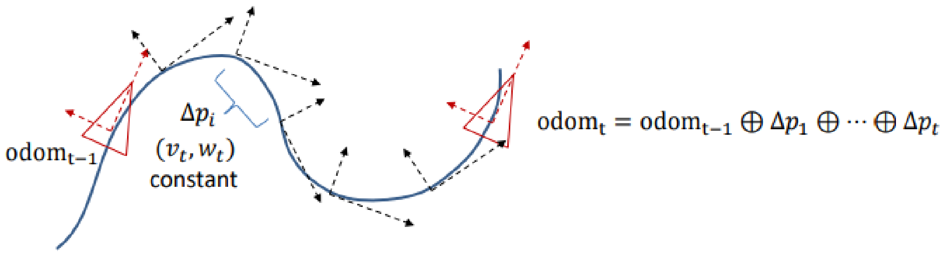

$odom_{t-1}$
Δp_i
(v_t, w_t)
constant
$odom_t = odom_{t-1} \oplus \Delta p_1 \oplus \cdots \oplus \Delta p_t$

Fig. 2: Example of composition of poses based on odometry.

Note that between the two odometry poses provided by the robotic base, there have been a series of pose increments computed by said firmware.

The **odometry motion model** consists of the utilization of such information that, although technically being a measurement rather than a control, will be treated as a control command to simplify the modeling. Thus, the odometry commands take the form of:

$$u_t = f(odom_t, odom_{t-1}) = \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta \theta \end{bmatrix}$$

being $odom_t$ and $odom_{t-1}$ measurements taken as control and computed from the odometry at time instants t and $t-1$.

We will implement this motion model in two different forms:

- Analytical form, where the motion command is an increment: $u_t = [\Delta x_t, \Delta y_t, \Delta \theta_t]^T$
- Sample form, where it is a combination of a rotation, motion in straight line, and rotation: $u_t = [\theta_1, d, \theta_2]^T$

Usage: We said that the ***velocity motion model*** is mainly used for motion planning, where the details of the robot's movement are of importance and odometry information is not available (*e.g.* no wheel encoders are available). The ***odometry model*** is more suitable to keep track and estimate the robot pose, it is more accurate, but measurements are available only after the motion is completed (or measured).

In [1]:
```
%matplotlib widget

# IMPORTS
import numpy as np
from numpy import random
import matplotlib.pyplot as plt
from scipy import stats
from IPython.display import display, clear_output
import time

import sys
sys.path.append("..")
from utils.DrawRobot import DrawRobot
from utils.PlotEllipse import PlotEllipse
from utils.pause import pause
from utils.Jacobians import J1, J2
from utils.tcomp import tcomp
```

OPTIONAL

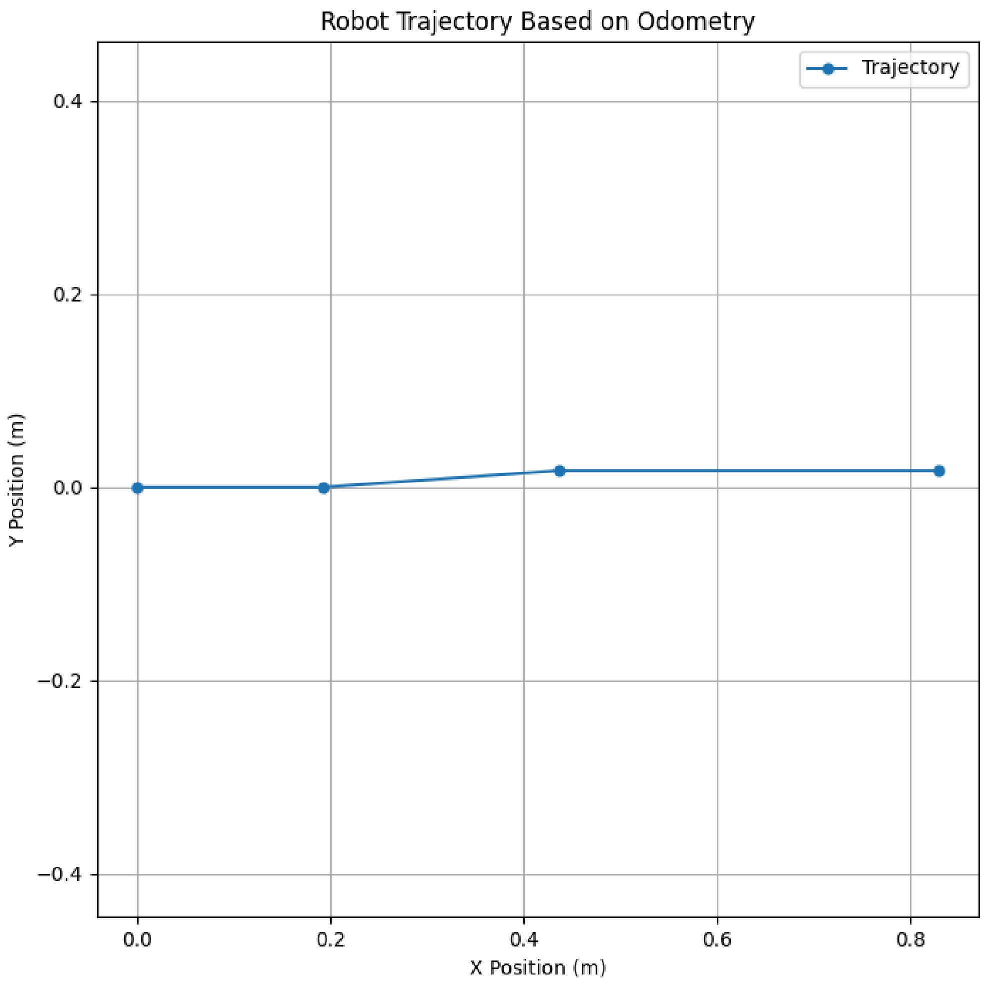
Let's compute an odometry pose as the robot base firmware does! Implement a method that, given a number of pulses detected in both wheels, computes the angles that the wheels turned and the resultant angular velocities. Then, implement a second one that retrieves the robot pose increment from those velocities, given a time increment Δt. Finally, given a vector of pulses detected from each wheel, compute their respective pose increments, and provide the final odometry pose.

In [50]:
```
def initialize_odometry(wheel_radius, wheel_base, pulses_per):
    pulses_to_rad = (2 * np.pi) / pulses_per
    return wheel_radius, wheel_base, pulses_to_rad

def compute_wheel_angles(left_pulses, right_pulses, pulses_to_rad):
    theta_left = left_pulses * pulses_to_rad
    theta_right = right_pulses * pulses_to_rad
    return theta_left, theta_right

def compute_pose_increment(left_angle, right_angle, wheel_radius, wheel_base, dt
    left_velocity = (wheel_radius * left_angle) / dt
    right_velocity = (wheel_radius * right_angle) / dt
    average_velocity = (left_velocity + right_velocity) / 2
    angular_velocity = (right_velocity - left_velocity) / wheel_base
    delta_x = average_velocity * dt
    delta_y = 0
    delta_theta = angular_velocity * dt

    return delta_x, delta_y, delta_theta


def compute_odometry(left_pulses, right_pulses, dt, wheel_radius, wheel_base, pu
    x, y, theta = 0, 0, 0
    trajectory = [(x, y)]
    for left_pulse, right_pulse in zip(left_pulses, right_pulses):
        left_angle, right_angle = compute_wheel_angles(left_pulse, right_pulse,
        delta_x, delta_y, delta_theta = compute_pose_increment(left_angle, right
        x += delta_x * np.cos(theta) - delta_y * np.sin(theta)
        y += delta_x * np.sin(theta) + delta_y * np.cos(theta)
        theta += delta_theta
        trajectory.append((x, y))

    return x, y, theta, trajectory
```

In [51]:

```
wheel_radius = 0.1
wheel_base = 0.5
pulses_per_rev = 360
dt = 1.0

wheel_radius, wheel_base, pulses_to_rad = initialize_odometry(wheel_radius, whee

left_pulses_vec = [100, 150, 200]
right_pulses_vec = [120, 130, 250]

# Compute odometry
final_x, final_y, final_theta, trajectory = compute_odometry(left_pulses_vec, ri

# Plotting the trajectory
trajectory = np.array(trajectory)
plt.figure(figsize=(8, 8))
plt.plot(trajectory[:, 0], trajectory[:, 1], marker='o', markersize=5, label='Tr
plt.title('Robot Trajectory Based on Odometry')
plt.xlabel('X Position (m)')
plt.ylabel('Y Position (m)')
plt.axis('equal')
plt.grid()
plt.legend()
plt.show()

final_position = (final_x, final_y, final_theta)
print(f"Final Position: {final_position}")
```

Figure



```
Final Position: (np.float64(0.8284361798899982), np.float64(0.01704472197982342),
0.17453292519943286)
```

END OF OPTIONAL PART

3.3.1 Analytic form

Just as we did in chapter 3.1, the analytic form of the odometry motion model uses the composition of poses to model the robot's movement, providing only a notion of how much the pose has changed, not how did it get there.

As with the *velocity model*, the odometry one uses a gaussian distribution to represent the **robot pose**, so $x_t \sim (\overline{x}_t, \Sigma_{x_t})$, being its mean and covariance computed as:

- **Mean:**

$$\overline{x}_t = g(\overline{x}_{t-1}, \overline{u}_t) = \overline{x}_{t-1} \oplus \overline{u}_t$$

where $u_t = [\Delta x_t, \Delta y_t, \Delta \theta_t]^T$, so:

$$g(\overline{x}_{t-1}, \overline{u}_t) = \begin{bmatrix} x_1 + \Delta x \cos\theta_1 - \Delta y \sin\theta_1 \\ y_1 + \Delta x \sin\theta_1 - \Delta y \cos\theta_1 \\ \theta_1 + \Delta\theta \end{bmatrix}$$

- **Covariance:**

Since the motion model $\overline{x}_t = g(\overline{x}_{t-1}, \overline{u}_t)$ applies a not linear transformation to compute $\overline{x}_t$, the result is not a Gaussian distribution, but it can be approximated. To approximate the covariance, we linearize said transformation and compute the covariance as:

$$\Sigma_{x_t} \approx \frac{\partial g}{\partial x_{t-1}, u_t} \begin{bmatrix} \Sigma_{x_{t-1}} & 0_{3x3} \\ 0_{3x3} & \Sigma_{u_t} \end{bmatrix} \frac{\partial g}{\partial x_{t-1}, u_t}$$

This can be simplified and expressed as:

$$\Sigma_{x_t} \approx \frac{\partial g}{\partial x_{t-1}} \cdot \Sigma_{x_{t-1}} \cdot \frac{\partial g}{\partial x_{t-1}}^T + \frac{\partial g}{\partial u_t} \cdot \Sigma_{u_t} \cdot \frac{\partial g}{\partial u_t}^T$$

where $\partial g / \partial x_{t-1}$ and $\partial g / \partial u_t$ are the jacobians of our motion model evaluated at the previous pose x_{t-1} and the current command u_t:

$$\frac{\partial g}{\partial x_{t-1}} = \frac{\partial g}{\partial \{x_{t-1}, y_{t-1}, \theta_{t-1}\}} = \begin{bmatrix} 1 & 0 & -\Delta x_t \sin\theta_{k-1} - \Delta y_t \cos\theta_{t-1} \\ 0 & 1 & \Delta x_t \cos\theta_{t-1} - \Delta y_t \sin\theta_{t-1} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\frac{\partial g}{\partial u_t} = \begin{bmatrix} \cos\theta_{t-1} & -\sin\theta_{t-1} & 0 \\ \sin\theta_{t-1} & \cos\theta_{t-1} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Recall how the Jacobian matrix of a vectorial function $f : \mathbb{R}^n -> f : \mathbb{R}^m$ is copmuted as:

$$\frac{\partial \{f_1, \dots, f_m\}}{\partial \{x_1, \dots, x_n\}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Jacobians are cool! They permit us to:

- Approximate the covariance matrix in the new pose by linearizing the composition function around the previous pose and the current control action by using the Taylor series expansion.
- Express two uncertainties given in two different reference systems (the world one of the previous pose and the local robot one of the motion command) in the same reference system so they can be added.
- If needed, they adapt covariance matrices so they share the same dimensions and they can be added.

Finally, the covariance matrix of this movement (Σ_{u_t}) is defined as seen below. Typically, it is constant during robot motion, although the *amount of motion* (travelled distance and turned angle) could be used to parametrize it. We will work with its constant version:

$$\Sigma_{u_t} = \begin{bmatrix} \sigma^2_{\Delta x} & 0 & 0 \\ 0 & \sigma^2_{\Delta y} & 0 \\ 0 & 0 & \sigma^2_{\Delta \theta} \end{bmatrix}$$

ASSIGNMENT 1: The model in action

Similarly to the assignment 3.1, we'll move a robot along a 8-by-8 square (in meters), in increments of 2m. In this case you have to complete:

- The `step()` method to compute:
  - the new expected pose (`self.pose`),
  - the new true pose x_t (ground-truth `self.true_pose`) after adding some noise using `stats.multivariate_normal.rvs()` to the movement command u according to `Q` (which represents Σ_{u_t}),
  - and to update the uncertainty about the robot position in `self.P` (covariance matrix Σ_{x_t}). Note that the methods `J1()` and `J2()` already implement $\partial g / \partial x_{t-1}$ and $\partial g / \partial u_t$ for you, you just have to call them with the right input parameters.
- The `draw()` method to plot:
  - the uncertainty of the pose as an ellipse centered at the expected pose, and
  - the true position (ground-truth).

We are going to consider the following motion covariance matrix (it is already coded for you):

$$\Sigma_{u_t} = \begin{bmatrix} 0.04 & 0 & 0 \\ 0 & 0.04 & 0 \\ 0 & 0 & 0.01 \end{bmatrix}$$

Example

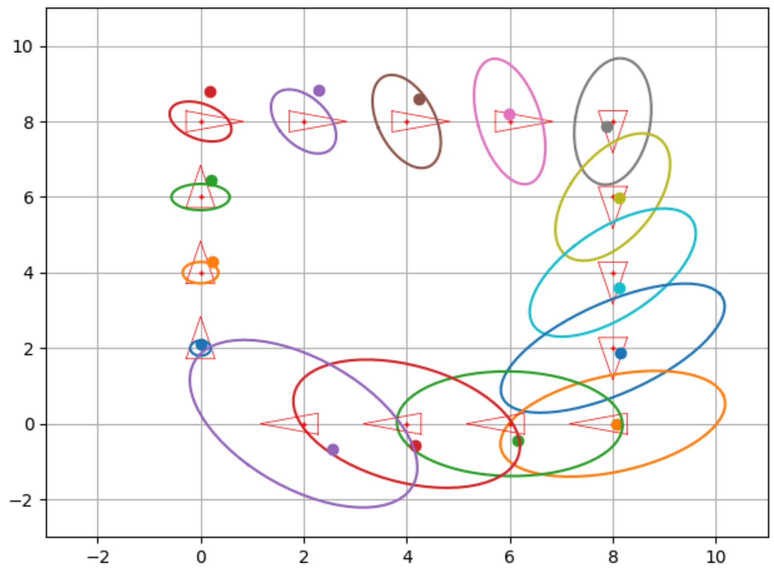


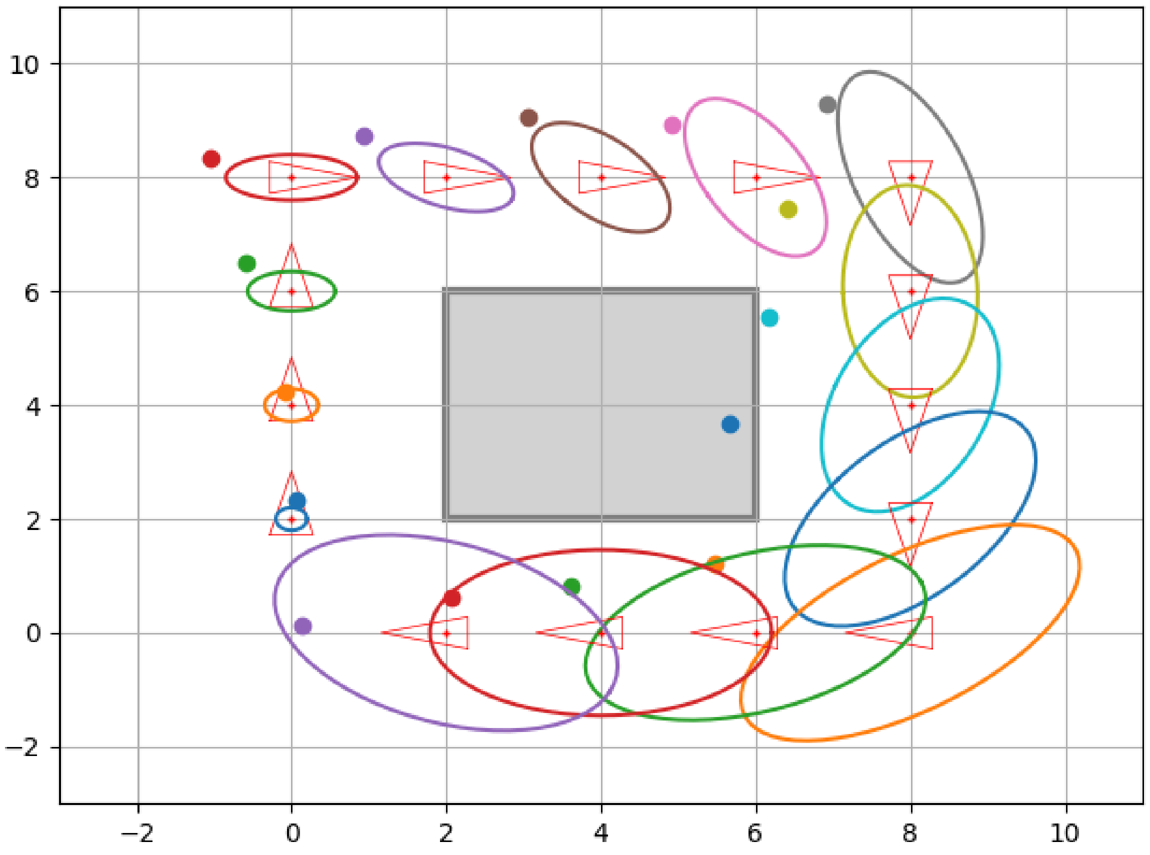
Fig. 2: Movement of a robot using odometry commands. Representing the expected pose (in red), the true pose (as dots) and the confidence ellipse.

In [11]:
```
class Robot():
    """ Simulation of a robot base
    
        Attrs:
            pose: Expected pose of the robot
            P: Covariance of the current pose
            true_pose: Real pose of the robot (affected by noise)
            Q: Covariance of the movement
    """
    def __init__(self, x, P, Q):
        self.pose = x
        self.P = P
        self.true_pose = self.pose
        self.Q = Q

    def step(self, u):
        # Update expected pose
        prev_pose = self.pose
        self.pose = tcomp(self.pose, u)

        # Generate true pose with noise
        noisy_u = np.vstack(stats.multivariate_normal.rvs(u.flatten(), self.Q))
        self.true_pose = tcomp(self.true_pose, noisy_u)

        # Update covariance
        JacF_x = J1(prev_pose, u)  # Ensure J1 returns a Jacobian
        JacF_u = J2(prev_pose, u)  # Ensure J2 returns a Jacobian

        self.P = (
            (JacF_x @ self.P @ JacF_x.T) + (JacF_u @ self.Q @ JacF_u.T)
        )

    def draw(self, fig, ax):
        DrawRobot(fig, ax, self.pose)
        el = PlotEllipse(fig, ax, self.pose, self.P)
        ax.plot(self.true_pose[0], self.true_pose[1], 'o', color=el[0].get_color
```

You can use the following demo to **try your new `Robot()` class**.

In [12]:
```
def demo_odometry_commands_analytical(robot):
    # MATPLOTLIB
    fig, ax = plt.subplots()
    ax.set_xlim([-3, 11])
    ax.set_ylim([-3, 11])
    plt.ion()
    plt.grid()
    plt.fill([2, 2, 6, 6],[2, 6, 6, 2],facecolor='lightgray', edgecolor='gray',
    plt.tight_layout()
    fig.canvas.draw()

    # MOVEMENT PARAMETERS
    nSteps = 15
    ang = -np.pi/2 # angle to turn in corners
    u = np.vstack((2., 0., 0.))

    # MAIN LOOP
    for i in range(nSteps):
        # change angle on corners
        if i % 4 == 3:
            u[2, 0] = ang

        #Update positions
        robot.step(u)

        # Restore angle iff changed
        if i % 4 == 3:
            u[2, 0] = 0

        # Draw every loop
        robot.draw(fig, ax)
        clear_output(wait=True)
        display(fig)
        time.sleep(0.3)

    plt.close()
```

In [13]:
```
x = np.vstack([0., 0., np.pi/2]) # pose inicial

# Probabilistic parameters
P = np.diag([0., 0., 0.])
Q = np.diag([0.04, 0.04, 0.01])

robot = Robot(x, P, Q)
demo_odometry_commands_analytical(robot)
```



Thinking about it (1)

Once you have completed this assignment regarding the analytical form of the odometry model, **answer the following questions**:

- Which is the difference between the $g(\cdot)$ function used here, and the one in the velocity model?

  In summary, $g(\cdot)$ in the velocity-based model computes future motion based on intended velocities, whereas in the odometry-based model, it computes future motion based on observed changes in the robot's position (via odometry).

- How many parameters compound the motion command u_t in this model?

  The motion command u_t in this model is composed of 3 parameters: $u_t = [x_t, y_t, \theta_t]$.

- Which is the role of the Jacobians $\partial g / \partial x_{t-1}$ and $\partial g / \partial u_t$?

  # The Jacobians $\partial g / \partial x_{t-1}$ and $\partial g / \partial u_t$ are used to update the covariance of the pose, by calculating the Jacobian of the $g(\cdot)$ function with respect to the previous pose and the motion command, respectively. We apreoximate the gaussian distribution of the new pose using the Jacobian becasue the function of the pose is not lineal.

- What happens if you modify the covariance matrix Σ_{u_t} modeling the uncertainty in the motion command u_t? Try different values and discuss the results.

  If the covariance matrix Σ_{u_t} is modified, the uncertainty in the motion command u_t will be different, and the covariance of the pose will be updated accordingly. If the *covariance of the motion command is increased, the covariance of the pose will also increase, and vice versa.*

3.3.2 Sample form

The analytical form used above, although useful for the probabilistic algorithms we will cover in this course, does not work well for sampling algorithms such as particle filters.

The reason being, if we generate random samples from the gaussian distributions as in the previous exercise, we will find some poses that are not feasible to the non-holonomic movement of a robot, i.e. they do not correspond to a velocity command (v, w) with noise.

The following *sample form* is a more realistic way to generate samples of the robot pose. In this case, the movement of the robot is modeled as a sequence of actions (see Fig 3):

1. **Turn** (θ_1): to face the destination point.
2. **Advance** (d): to arrive at the destination.
3. **Turn** (θ_2): to get to the desired angle.

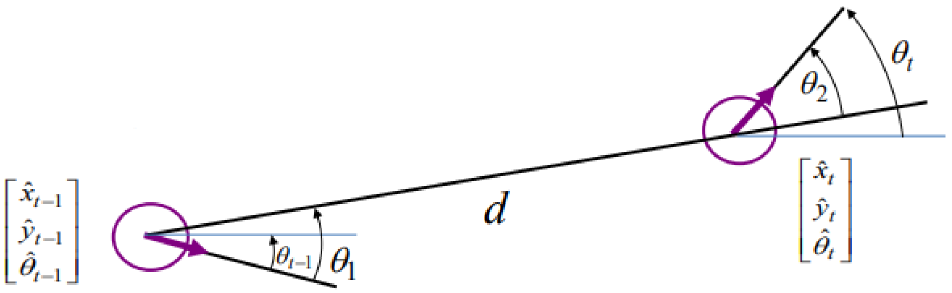

$\begin{bmatrix}\hat{x}_{t-1}\\ \hat{y}_{t-1}\\ \hat{\theta}_{t-1}\end{bmatrix}$
θ_{t-1}
θ_1
d
θ_2
θ_t
$\begin{bmatrix}\hat{x}_t\\ \hat{y}_t\\ \hat{\theta}_t\end{bmatrix}$

Fig. 3: Movement of a robot using odometry commands in sampling form.

So this type of order is expressed as:

$$u_t = \begin{bmatrix}\theta_1\\ d\\ \theta_2\end{bmatrix}$$

It can easily be generated from odometry poses $[\hat{x}_t, \hat{y}_t, \hat{\theta}_t]^T$ and $[\hat{x}_{t-1}, \hat{y}_{t-1}, \hat{\theta}_{t-1}]^T$ given the following equations:

$$\theta_1 = atan2(\hat{y}_t - \hat{y}_{t-1}, \hat{x}_t - \hat{x}_{t-1}) - \hat{\theta}_{t-1}$$

$$d = \sqrt{(\hat{y}_t - \hat{y}_{t-1})^2 + (\hat{x}_t - \hat{x}_{t-1})^2}$$

$$\theta_2 = \hat{\theta}_t - \hat{\theta}_{t-1} - \theta_1$$

This model works by considering a set of possible robot's locations $\{x^i_{t-1}\}, i = 1, \dots, n$ and, when a new odometry measurement comes, the motion command that produced it u_t is retrieved. Then, the motion model $x_t = g(x_{t-1}, u)$ is used to obtain $\{x^i_t\}, i = 1, \dots, n$ where each particle x^i_{t-1} is moved according to a perturbed version of u_t.

Note: the hat ^ indicates values in the robot's internal coordinate system, which may not match the world reference system.

ASSIGNMENT 2: Implementing the sampling form

Complete the following cells to experience the motion of a robot using the sampling form of the odometry model. For that:

1. Implement a function that, given the previously mentioned $[\hat{x}_t, \hat{y}_t, \hat{\theta}_t]^T$ and $[\hat{x}_{t-1}, \hat{y}_{t-1}, \hat{\theta}_{t-1}]^T$ generates an order $u_t = [\theta_1, d, \theta_2]^T$

In [31]:
```python
def generate_move(pose_now, pose_old):
    diff = pose_now - pose_old
    theta1 = np.arctan2(diff[1], diff[0]) - pose_old[2]
    d = np.sqrt(diff[0]**2 + diff[1]**2)
    theta2 = diff[2] - theta1
    return np.vstack((theta1, d, theta2))
```

Try such function with the code cell below:

In [33]:
```python
generate_move(np.vstack([0., 0., 0.]), np.vstack([1., 1., np.pi/2]))
```

Out[33]:
```
array([[-3.92699082],
       [ 1.41421356],
       [ 2.35619449]])
```

Expected output for the commented example:

```
array([[-3.92699082],
       [ 1.41421356],
       [ 2.35619449]])
```

2. Using the resulting control action $u_t = [\hat{\theta}_1, \hat{d}, \hat{\theta}_2]^T$ we can model its noise in the following way:

$$\theta_1 = \hat{\theta}_1 + \text{sample}\left(\alpha_0 \hat{\theta}_1^2 + \alpha_1 \hat{d}^2\right)$$

$$d = \hat{d} + \text{sample}\left(\alpha_2 \hat{d}^2 + \alpha_3\left(\hat{\theta}_1^2 + \hat{d}^2\right)\right)$$

$$\theta_2 = \hat{\theta}_2 + \text{sample}\left(\alpha_0 \hat{\theta}_2^2 + \alpha_1 \hat{d}^2\right)$$

Where $sample(b)$ generates a random value from a distribution $N(0, b)$. The vector $\alpha = [\alpha_0, \ldots, \alpha_3]$ (`a` in the code), models the robot's intrinsic noise. As before, we are taking the control action and adding some noise to account for its uncertainty.

In this case, the motion model $x_t = g(x_{t-1}, u_t)$ sets that the pose of the robot at the end of the movement is computed as follows:

$$x_t = x_{t-1} + d\cos(\theta_{t-1} + \theta_1)$$

$$y_t = y_{t-1} + d\sin(\theta_{t-1} + \theta_1)$$

$$\theta_t = \theta_{t-1} + \theta_1 + \theta_2$$

Complete the `step()` and `draw()` methods to:

- Update the expected robot pose (`self.pose`) and generate new samples. The number of samples is set by `n_samples` , and `self.samples` is in charge of storing such samples. Each sample can be interpreted as one possible pose reached by the robot.
- Draw the true pose of the robot (without angle) as a cloud of particles (samples of possible points which the robot can be at). Play a bit with different values of `a` . To improve this visualization the robot will move in increments of 0.5 and we are going to plot the particles each 4 increments.

Example

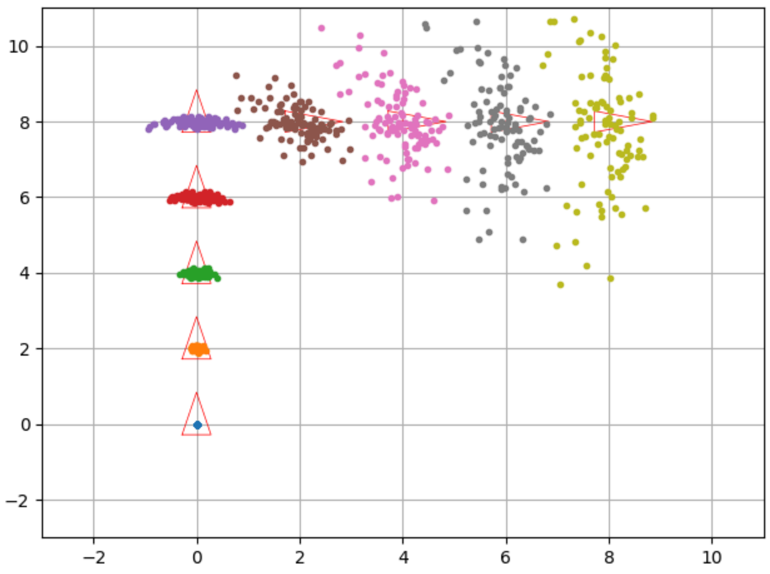


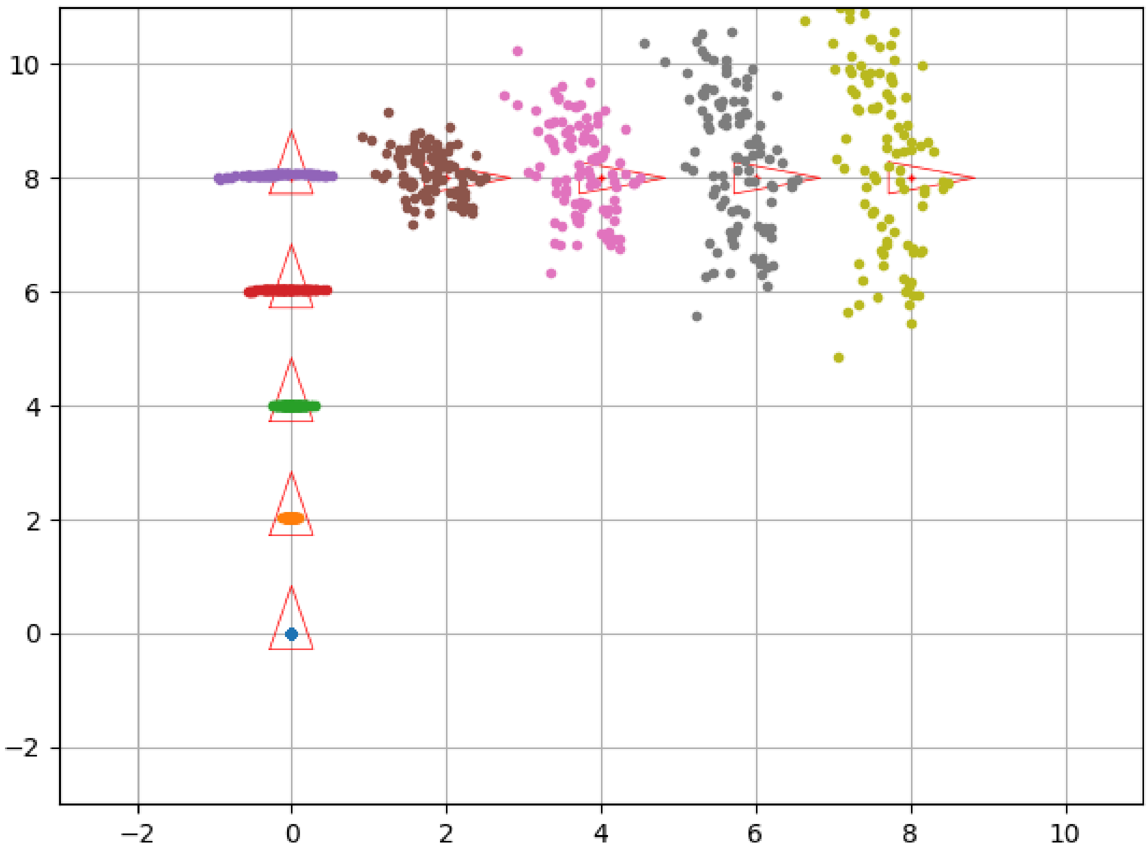
Fig. 1: Movement of a robot using odometry commands in sampling form. Representing the expected pose (in red) and the samples (as clouds of dots)

In [34]:
```
class SampledRobot(object):
    def __init__(self, mean, a, n_samples):
        self.pose = mean
        self.a = a
        self.samples = np.tile(mean, n_samples)

    def step(self, u):
        # TODO Update pose
        ang = self.pose[2, 0] + u[0, 0]
        self.pose[0, 0] += u[1, 0]*np.cos(ang)
        self.pose[1, 0] += u[1, 0]*np.sin(ang)
        self.pose[2, 0] = ang+u[2, 0]

        # TODO Generate new samples
        sample = lambda b: stats.norm(loc=0, scale=b).rvs(size=self.samples.shap

        u2 = u**2
        noisy_u = u + np.vstack((
            sample(self.a[0]*u[0, 0]**2 + self.a[1]*u[1, 0]**2),
            sample(self.a[2]*u[1, 0]**2 + self.a[3]*(u[0, 0]**2 + u[1, 0]**2)),
            sample(self.a[0]*u[2, 0]**2 + self.a[1]*u[1, 0]**2)
        ))

        # TODO Update particles (robots) poses
        ang = self.samples[2, :] + noisy_u[0, :]

        self.samples[0, :] += noisy_u[1, 0]*np.cos(ang)
        self.samples[1, :] += noisy_u[1, 0]*np.sin(ang)
        self.samples[2, :] = ang+noisy_u[2, 0]

    def draw(self, fig, ax):
        DrawRobot(fig, ax, self.pose)
        ax.plot(self.samples[0, :], self.samples[1, :], '.')
```

Run the following demo to **test your code**:

In [35]:
```
def demo_odometry_commands_sample(robot):
    # PARAMETERS
    inc = .5
    show_each = 4
    limit_iterations = 32

    # MATPLOTLIB
    fig, ax = plt.subplots()
    ax.set_xlim([-3, 11])
    ax.set_ylim([-3, 11])
    plt.ion()
    plt.grid()
    plt.tight_layout()

    # MAIN LOOP
    robot.draw(fig, ax)
    inc_pose = np.vstack((0., inc, 0.))

    for i in range(limit_iterations):
        if i == 16:
            inc_pose[0, 0] = inc
            inc_pose[1, 0] = 0
            inc_pose[2, 0] = -np.pi/2

        u = generate_move(robot.pose+inc_pose, robot.pose)

        robot.step(u)

        if i == 16:
            inc_pose[2, 0] = 0

        if i % show_each == show_each-1:
            robot.draw(fig, ax)
            clear_output(wait=True)
            display(fig)
            time.sleep(0.1)

    plt.close()
```

In [37]:
```
# RUN
n_particles = 100
a = np.array([.07, .07, .03, .05])
x = np.vstack((0., 0., np.pi/2))

robot = SampledRobot(x, a, n_particles)
demo_odometry_commands_sample(robot)
```



Thinking about it (2)

Now you are an expert in the sample form of the odometry motion model! **Answer the following questions**:

- Which is the effect of modifying the robot's intrinsic noise α (`a` in the code)?

  The answer is that the robot will have more or less noise in its movements. If we increase alpha, the robot will have more noise in its movements, and if we decrease it, the robot will have less noise in its movements.

- How many parameters compound the motion command u_t in this model?

  3 parameters.

- After moving the robot a sufficient number of times, what shape does the distribution of samples take?

  The banana shape.