3.3. Odometry-based motion model

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*.

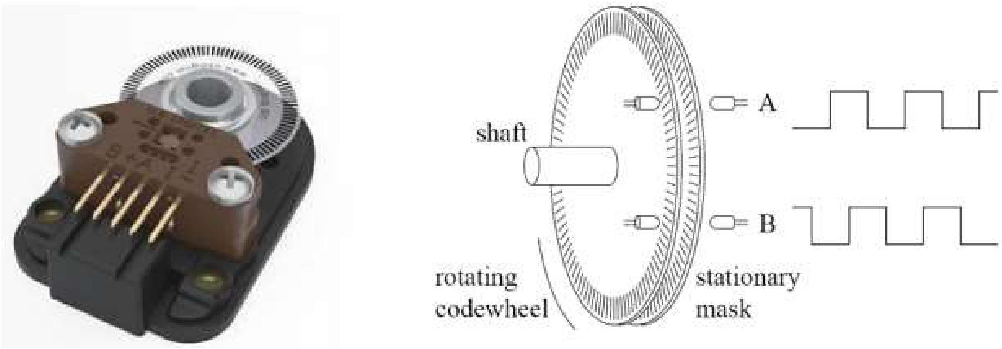


Fig. 1: Example of a wheel encoder used to sum pulses and compute the robot pose.

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:

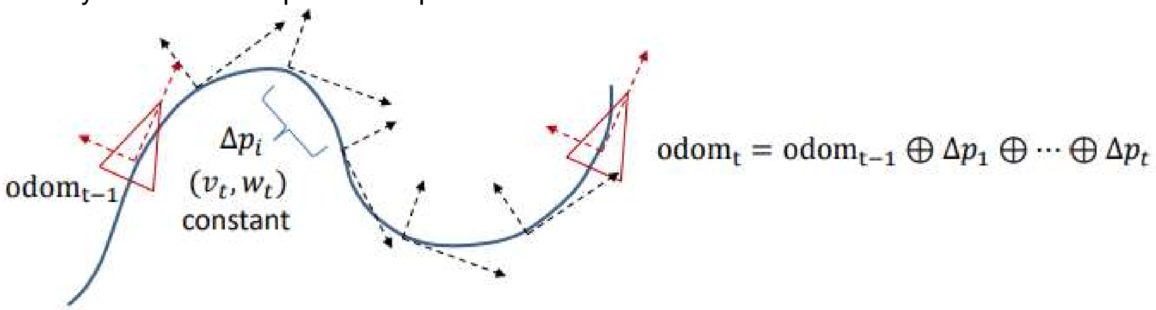


Fig. 2: Example of composition of poses based on odometry.

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: $u_t = [\theta_1, d, \theta_2]^T$

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.

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:**

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

  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:

$$\frac{\partial g}{\partial x_{k-1}} = \begin{bmatrix} 1 & 0 & -\Delta x_k \sin\theta_{k-1} - \Delta y_k \cos\theta_{k-1} \\ 0 & 1 & \Delta x_k \cos\theta_{k-1} - \Delta y_k \sin\theta_{k-1} \\ 0 & 0 & 1 \end{bmatrix} \qquad \frac{\partial g}{\partial u_k} = \begin{bmatrix} \cos\theta_{k-1} & -\sin \\ \sin\theta_{k-1} & \cos \\ 0 & (\end{bmatrix}$$

and 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

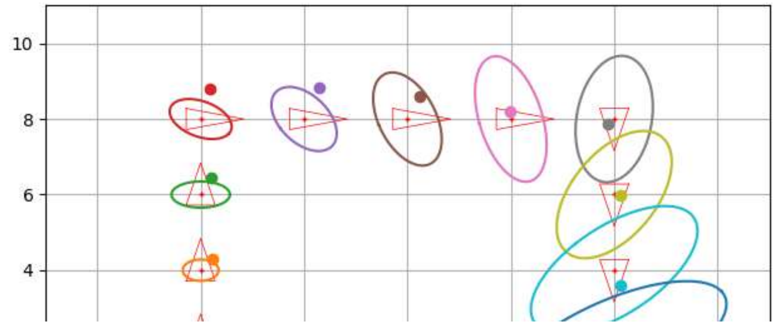
Similarly to the assignment 3.1, we'll move a robot along a 8-by-8 square (in meters), in increments of 2m. In this case you have to complete:

- 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()` (https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.multivariate_normal.l](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.multivariate_normal.l) 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

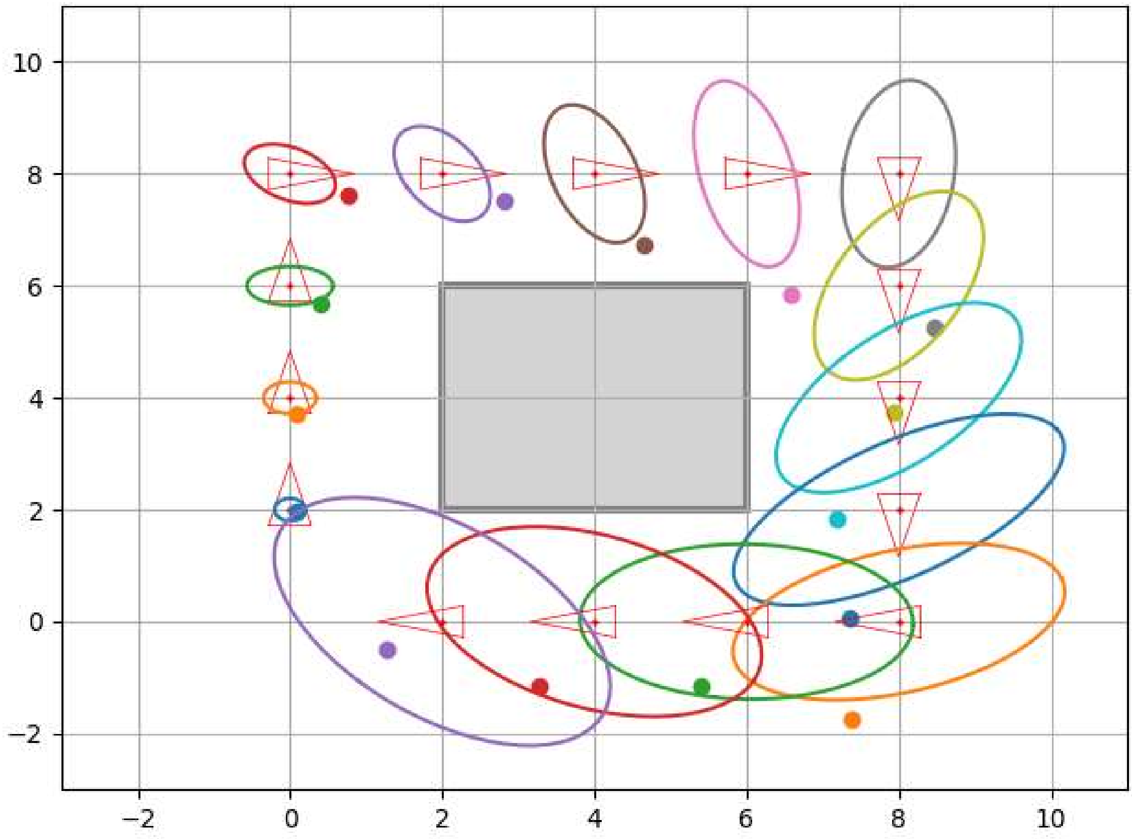


In [2]:
```python
class Robot():
    """ Simulation of a robot base

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

        # TODO Generate true pose
        noisy_u = np.vstack(stats.multivariate_normal.rvs(u.flatten(), self.
        #np.random.multivariate_normal(u.flatten(),cov)

        self.true_pose = tcomp(self.true_pose, noisy_u)

        # TODO Update covariance
        JacF_x = J1(self.pose, u)
        JacF_u = J2(self.pose, u)

        # Similar a pr3.2. Formula de arriba.
        self.P = (
            (JacF_x @ self.P @ np.transpose(JacF_x)) +
                (JacF_u @ self.Q @ np.transpose(JacF_u))
        )
    # Similar a pr 3.2
    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_c
```

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

In [3]:
```
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='gra
    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 [10]:
```
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 the velocity model, g(.) is responsible for transitioning from the previous pose to the new pose. The control command u_t includes linear and non linear velocities. This results in a continuous update of the robot's position and orientation as it moves. - In the odometry model, the g(.) function also transitions from the previous pose to the new pose. However, it takes the previous pose x_{t-1} and the pose increment as input. The pose increment u_t represents incremental changes in position and orientation. The motion model updates the robot's pose based on these incremental changes and it leads to a discrete update of the robot's position and orientation.

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

  Three parameters. These parameters represent the incremental changes in the robot's pose from the previous time step to the current time step. 1. Delta{x_t}: The change in the x-coordinate of the robot's position. 2. Delta{y_t}: The change in the y-coordinate of the robot's position. 3. Delta{theta_t}: The change in the robot's orientation.

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

  Jacobians are used to update the covariance matrix of the robot's pose through the covariance update equation above. They show the relationship between the previous pose, the motion command, and the pose increment. They help incorporate uncertainty and account for how changes in pose and motion command affect the uncertainty in the robot's estimated pose.

- 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.

  When you increase (decrease) the values in Sigma_{u_t}, it means you are introducing more (less) uncertainty into the motion commands. The covariance matrix Sigma_{x_t} will also increase because the uncertainty from the motion commands propagates to the pose estimation. - If you increase (decrease) Delta{x_t} we will get more (less) uncertainty in the X-axis. The elipse in the X-axis will get bigger (smaller). - If you increase (decrease) Delta{y_t} we will get more (less) uncertainty in the Y-axis. The elipse in the Y-axis will get bigger (smaller). - If you increase (decrease) Delta{theta_t} we will get more (less) uncertainty in the robot rotation.

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.

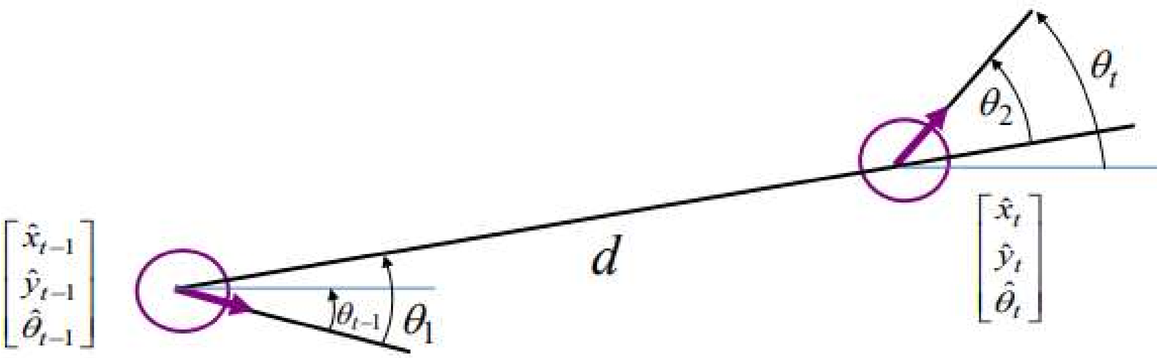


Fig. 3: Movement of a robot using odometry commands in sampling form.

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:

$$\theta_1 = atan2(\hat{y}_t - \hat{y}_{t-1}, \hat{x}_t - \hat{x}_{t-1}) - \hat{\theta}_{t-1}$$
$$d = \sqrt{(\hat{y}_t - \hat{y}_{t-1})^2 + (\hat{x}_t - \hat{x}_{t-1})^2}$$
$$\theta_2 = \hat{\theta}_t - \hat{\theta}_{t-1} - \theta_1$$

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 [7]:
```
def generate_move(pose_now, pose_old):
    diff = pose_now - pose_old
    theta1 = np.arctan2(pose_now[1] - pose_old[1], pose_now[0] - pose_old[0]
    d = np.sqrt((pose_now[0] - pose_old[0])**2 + (pose_now[1] - pose_old[1])
    theta2 = pose_now[2] - pose_old[2] - theta1
    return np.vstack((theta1, d, theta2))
```

Try such function with the code cell below:

In [8]:
```
generate_move(np.vstack([0., 0., 0.]), np.vstack([1., 1., np.pi/2]))
```

Out[8]:
```
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.

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

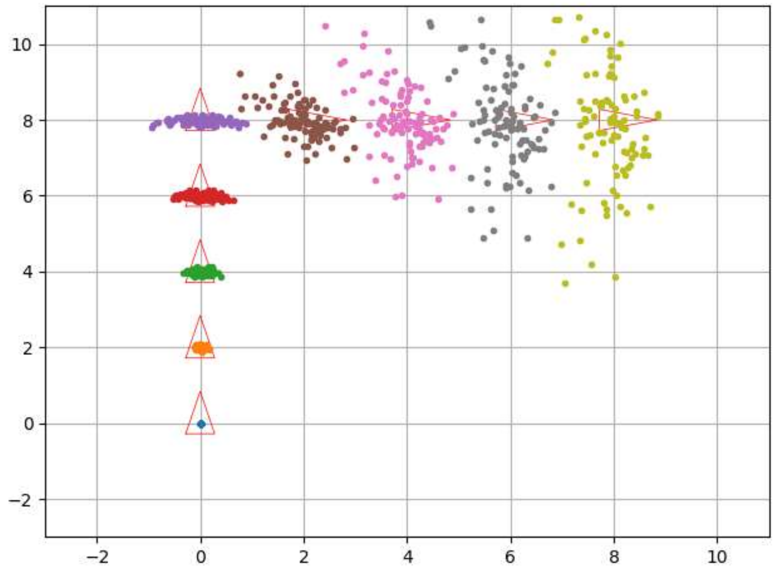


Fig. 1: Movement of a robot using odometry commands in sampling form.

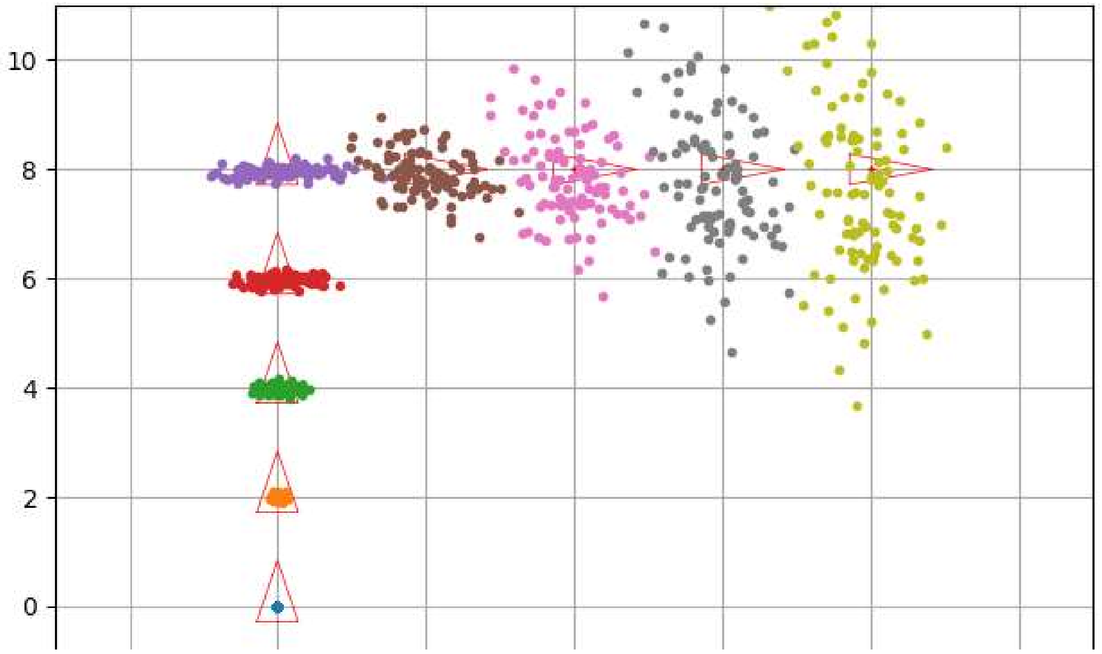
In [42]:
```
class SampledRobot(object):
    def __init__(self, mean, a, n_samples):
        self.pose = mean
        self.a = a
        self.samples = np.tile(mean, n_samples)

    def step(self, u):
        # TODO Update pose
        ang = self.pose[2, 0] + u[0, 0] # theta_t-1 + theta_1
        self.pose[0, 0] += u[1, 0] * np.cos(ang) # x_t-1 + d * cos(theta_t-1
        self.pose[1, 0] += u[1, 0] * np.sin(ang) # y_t-1 + d * sin(theta_t-1
        self.pose[2, 0] = ang + u[2, 0] # theta_t-1 + theta_1 + theta_2
        # u se trata como un vector de 3x1, donde u[0] = theta_1, u[1] = d,

        # TODO Generate new samples
        sample = lambda b: stats.norm(loc=0, scale=b).rvs(size=self.samples.
        # sample es una funcion que genera una muestra de una distribucion n
        u2 = u**2

        noisy_u = u + np.vstack((
            sample(self.a[0] * u2[0, 0] + self.a[1] * u2[1, 0]), # theta_1
            sample(self.a[2] * u2[1, 0] + self.a[3] * (u2[0, 0] + u2[1, 0]))
            sample(self.a[0] * u2[2, 0] + self.a[1] * u2[1, 0]) # theta_2
        ))

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

        self.samples[0, :] += noisy_u[1, :] * np.cos(ang) # x_t-1 + d * cos(
        self.samples[1, :] += noisy_u[1, :] * np.sin(ang) # y_t-1 + d * sin(
        self.samples[2, :] = ang + noisy_u[2, :] # theta_t-1 + theta_1 + the

    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 [43]:
```python
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 [44]:

```
# 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)?

  Higher values lead to more noise in the control command, resulting in a wider spread of samples and increased uncertainty in the robot's pose. However, lower values reduce the noise and make the samples more concentrated, reducing the uncertainty in the robot's pose during its motion.

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

  Three parameters. It is represented as a vector: u = [theta_1, d, theta_2]. - theta_1: first rotation angle, which corresponds to the turn the robot makes to face the destination point. - d: linear distance, which corresponds to the movement of the robot to reach the destination. - theta_2: second rotation angle, which corresponds to the turn the robot makes to get to the desired final angle.

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

  It is a banana shape. The distribution represents the uncertainty in the robot's pose after multiple movements, considering the noise in the control commands and the robot's motion.