3.1 Motion through pose composition

A fundamental aspect of the development of mobile robots is the motion itself. In an idyllic world, motion commands are sent to the robot locomotion system, which perfectly executes them and drives the robot to a desired location. However, this is not a trivial matter, as many sources of motion error appear:

- wheel slippage,
- inaccurate calibration,
- limited resolution during integration (time increments, measurement resolution), or
- unequal floor, among others.

These factors introduce uncertainty in the robot motion. Additionally, other constraints to the movement difficult its implementation. This particular chapter explores the concept of *robot's pose* and how we deal with it in a probabilistic context.

The pose itself can take multiple forms depending on the problem context:

- **2D location**: In a planar context we only need to a 2d vector $[x, y]^T$ to locate a robot against a point of reference, the origin $(0, 0)$.
- **2D pose**: In most cases involving mobile robots, the location alone is insufficient. We need an additional parameter known as orientation or *bearing*. Therefore, a robot's pose is usually expressed as $[x, y, \theta]^T$ (see Fig. 1). *In the rest of the book, we mostly refer to this one.*
- **3D pose**: Although we will only mention it in passing, for robotics applications in the 3D space, *i.e.* UAV or drones, not only a third axis z is added, but to handle the orientation in a 3D environment we need 3 components, *i.e.* roll, pitch and yaw. This course is centered around planar mobile robots so we will not use this one, nevertheless most methods could be adapted to 3D environments.

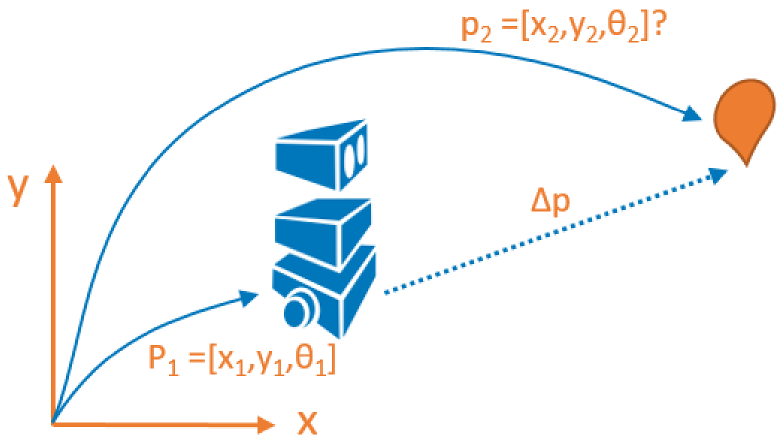


Fig. 1: Example of an initial 2D robot pose (p_1) and its resultant pose (p_2) after completing a motion (Δp).

In this chapter we will explore how to use the **composition of poses** to express poses in a certain reference system, while the next two chapters describe two probabilistic methods for dealing with the uncertainty inherent to robot motion, namely the **velocity-based** motion model and the **odometry-based** one.

In [1]:
```
%matplotlib widget

# IMPORTS

import numpy as np
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.tcomp import tcomp
```

OPTIONAL

In the Robot motion lecture, we started talking about *Differential drive* motion systems. Include as many cells as needed to introduce the background that you find interesting about it and some code illustrating some related aspect, for example, a code computing and plotting the *Instantaneus Center of Rotation (ICR)* according to a number of given parameters.

END OF OPTIONAL PART

3.1 Pose composition

The composition of posses is a tool that permits us to express the *final* pose of a robot in an arbitrary coordinate system. Given an initial pose p_1 and a pose differential Δp (pose increment), *i.e.* how much the robot has moved during an interval of time, the final pose p can be computed using the **composition of poses** function:

$$p_1 = \begin{bmatrix} x_1 \\ y_1 \\ \theta_1 \end{bmatrix}, \quad \Delta p = \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta \theta \end{bmatrix}$$

$$p = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} = p_1 \oplus \Delta p = \begin{bmatrix} x_1 + \Delta x \cos\theta_1 - \Delta y \sin\theta_1 \\ y_1 + \Delta x \sin\theta_1 + \Delta y \cos\theta_1 \\ \theta_1 + \Delta\theta \end{bmatrix}$$

The differential Δp, although we are using it as control in this exercise, normally is calculated given the robot's locomotion or sensed by the wheel encoders.

OPTIONAL

Implement your own methods to compute the composition of two poses, as well as the inverse composition. Include some examples of their utilization, also incorporating plots.

END OF OPTIONAL PART

ASSIGNMENT 1: Moving the robot by composing pose increments

Take a look at the `Robot()` class provided and its methods: the constructor, `step()` and `draw()`. Then, modify the main function in the next cell for the robot to describe a $8m \times 8m$ square path as seen in the figure below. You must take into account that:

- The robot starts in the bottom-left corner $(0, 0)$ heading north and
- moves at increments of $2m$ each step.
- Each 4 steps, it will turn right.

Example

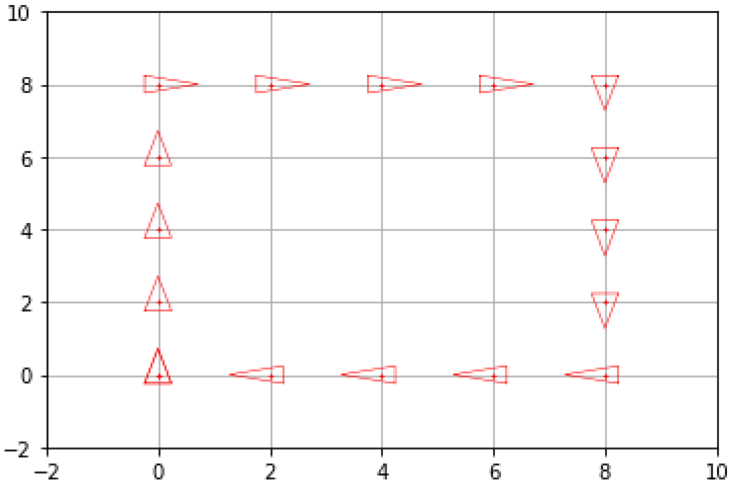


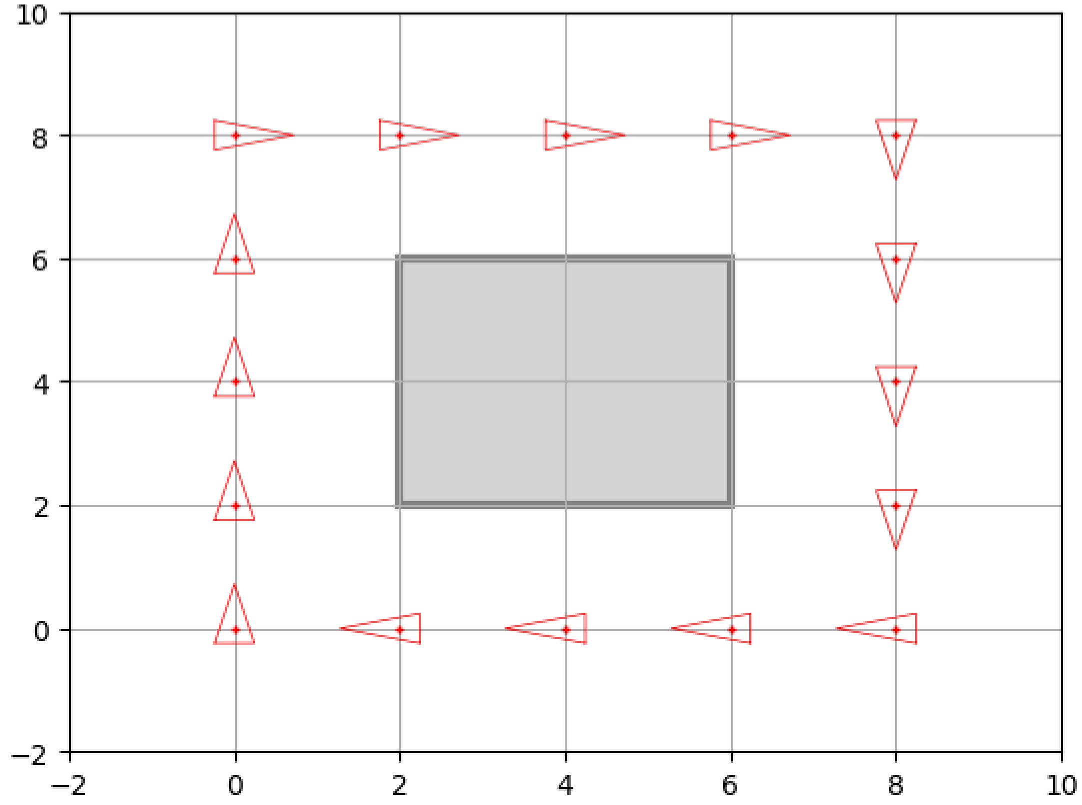
Fig. 2: Route of our robot.

In [2]:
```
class Robot():
    '''Mobile robot implementation

        Attr:
            pose: Expected position of the robot
    '''
    def __init__(self, mean):
        self.pose = mean

    def step(self, u):
        self.pose = tcomp(self.pose, u)

    def draw(self, fig, ax):
        DrawRobot(fig, ax, self.pose)
```

In [3]:
```
def main(robot):

    # PARAMETERS INITIALIZATION
    num_steps = 15 # Number of robot motions
    turning = 4  # Number of steps for turning
    u = np.vstack([2., 0., 0.]) # Motion command (pose increment)
    angle_inc = -np.pi/2 # Angle increment

    # VISUALIZATION
    fig, ax = plt.subplots()
    plt.ion()
    plt.draw()
    plt.xlim((-2, 10))
    plt.ylim((-2, 10))
    plt.fill([2, 2, 6, 6],[2, 6, 6, 2],facecolor='lightgray', edgeco

    plt.grid()
    robot.draw(fig, ax)

    # MAIN LOOP
    for step in range(1,num_steps+1):

        # Check if the robot has to move in straight line or also has
        # and accordingly set the third component (rotation) of the m
        if step%turning == 0:
            u[2] = angle_inc
        else:
            u[2] = 0.

        # Execute the motion command
        robot.step(u)

        # VISUALIZATION
        robot.draw(fig, ax)
        clear_output(wait=True)
        display(fig)
        time.sleep(0.1)

    plt.close()
```

Execute the following code cell to **try your code**. The resulting figure must be the same as Fig. 2.

In [4]:
```
# RUN
initial_pose = np.vstack([0., 0., np.pi/2])
robot = Robot(initial_pose)
main(robot)
```



3.2 Considering noise

In the previous case, the robot motion was error-free. This is overly optimistic as in a real use case the conditions of the environment are a huge source of uncertainty.

To take into consideration such uncertainty, we will model the movement of the robot as a (multidimensional) gaussian distribution $\Delta p \sim N(\mu_{\Delta p}, \Sigma_{\Delta p})$ where:

- The mean $\mu_{\Delta p}$ is still the pose differential in the previous exercise, that is Δp_{given}.
- The covariance $\Sigma_{\Delta p}$ is a 3×3 matrix, which defines the amount of error at each step (time interval).

ASSIGNMENT 2: Adding noise to the pose motion

Now, we are going to add a Gaussian noise to the motion, assuming that the incremental motion now follows the probability distribution:

$$\Delta p = N(\Delta p_{given}, \Sigma_{\Delta p}) \; with \; \Sigma_{\Delta p} = \begin{bmatrix} 0.04 & 0 & 0 \\ 0 & 0.04 & 0 \\ 0 & 0 & 0.01 \end{bmatrix} (\text{ units in } m^2 \text{ and } rad^2)$$

For doing that, complete the `NosyRobot()` class below, which is a child class of the previous `Robot()` one. Concretely, you have to:

- Complete this new class by adding some amount of noise to the movement (take a look at the `step()` method. *Hints:* [`np.vstack()` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.vstack.html)](https://docs.scipy.org/doc/numpy/reference/generated/numpy.vstack.html), [`stats.multivariate_normal.rvs()` (https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.multivariate_normal.htm](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.multivariate_normal.htm)
Remark that we have now two variables related to the robot pose:
  - `self.pose` , which represents the expected, *ideal* pose, and
  - `self.true_pose` , that stands for the actual pose after carrying out a noisy motion command.
- Along with the expected pose drawn in red (`self.pose`), in the `draw()` method plot the real pose of the robot (`self.true_pose`) in blue, which as commented is affected by noise.

Run the cell several times to see that the motion (and the path) is different each time. Try also with different values of the covariance matrix.

Example

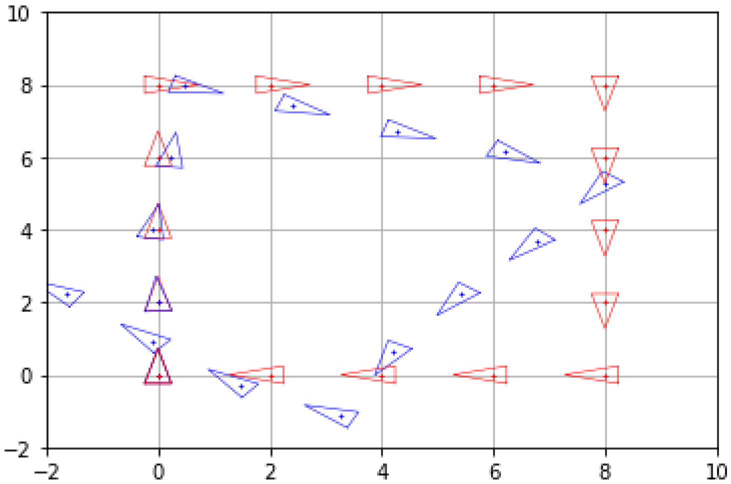
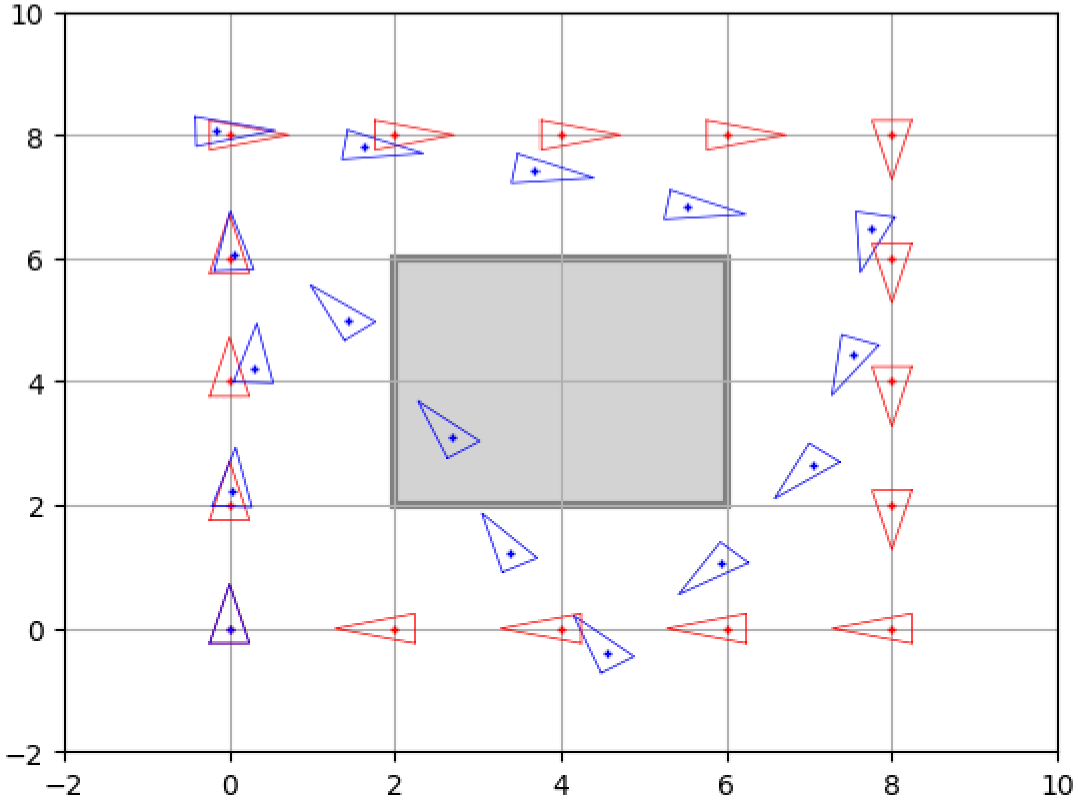


Fig. 3: Movement of our robot using pose compositions. Containing the expected poses (in red) and the true pose affected by noise (in blue)

In [5]:
```
class NoisyRobot(Robot):
    """Mobile robot implementation. It's motion has a set ammount of 

        Attr:
            pose: Inherited from Robot
            true_pose: Real robot pose, which has been affected by so
            covariance: Amount of error of each step.
    """
    def __init__(self, mean, covariance):
        super().__init__(mean)
        self.true_pose = mean
        self.covariance = covariance

    def step(self, step_increment):
        """Computes a single step of our noisy robot.

            super().step(...) updates the expected pose (without nois
            Generate a noisy increment based on step_increment and se
            Then this noisy increment is applied to self.true_pose
        """
        super().step(step_increment)
        true_step = stats.multivariate_normal.rvs(mean = step_increme
        self.true_pose = tcomp(self.true_pose, np.vstack(true_step))

    def draw(self, fig, ax):
        super().draw(fig, ax)
        DrawRobot(fig, ax, self.true_pose, color='blue')
```

In [8]:
```
# RUN
initial_pose = np.vstack([0., 0., np.pi/2])
cov = np.diag([0.04, 0.04, 0.01])
#cov = [[0.5,0.5,0],[0.5,0.5,0],[0,0,0]]

robot = NoisyRobot(initial_pose, cov)
main(robot)
```



Thinking about it (1)

Now that you are an expert in retrieving the pose of a robot after carrying out a motion command defined as a pose increment, **answer the following questions**:

- Why are the expected (red) and true (blue) poses different?

  Al haber introducido ruido al movimiento de la pose en la pose verdadera, la trayectoria del robot se ha visto afectada de cierta manera (pudiendo variar entre ejecuciones) que ha ocasionado una secuencia de pasos reales diferentes a los esperados sin ruido.

- In which scenario could they be the same?

  Serán exactamente iguales si los valores representados para la pose real fueran siempre la media de la distribución normal (lo cual no siempre se cumple, pero no es imposible que se dé puesto que depende de los valores de la matriz de covarianzas). Recordemos que en este caso con ruido, el incremento de pose sigue una normal centrada en el mismo valor que en el caso sin ruido, pero por haber incluido la matriz de covarianzas hace que no siempre se pueda obtener este valor, habiendo introducido incertidumbre en el proceso.

  Otra respuesta en el que ambos coincidan es que sea una matriz de ceros (trivial). Esta situación se podría interpretar como si no hubiera ruido.

- How affect the values in the covariance matrix $\Sigma_{\Delta p}$ the robot motion?

En primer lugar hemos de darnos cuenta que la matriz de covarianzas es una matriz diagonal. Esto quiere decir que la incertidumbre en alguna de sus dimensiones (eje X, eje Y más el ángulo que forman) no afecta a las demás.

A medida que estos valores de la diagonal se alejen progresivamente de cero (del orden de las décimas de unidad), estaremos introduciendo ruido que repercutirá en una incertidumbre mayor. Pequeños valores de ruido "casi no se notarán" pero grandes valores harán que la pose real tenga una trayectoria errática. Notar que los valores de ruido en el ángulo deben de ser inferiores al resto puesto que tiene una mayor repercusión en la trayectoria real.

3.2 Velocity-based motion model

In the remainder of this chapter we will describe two probabilistic motion models for planar movement: the **velocity motion model** and the **odometry motion model**, the former being the main topic of this section. Remember that when a movement command is given to a robot, there are different factors that affect such movement (*e.g.* wheel slippage, unequal floor, inaccurate calibration, etc.), adding uncertainty to the actual move done. This results in a need for characterizing the robot motion in *probabilistic terms*, that is:

$$p(x_t|u_t, x_{t-1})$$

being:

- x_t the robot pose at time instant t,
- u_t the motion command (also called control action) at t, and
- x_{t-1} the robot pose at the previous time instant $t-1$.

So basically this probability models the probability distribution over robot poses when executing the motion command u_t, having the robot the previous pose x_{t-1}. In other words, we are considering a function $g(\cdot)$ that performs $x_t = g(x_{t-1}, u_t)$ and outputs $x_t \sim p(x_t|u_t, x_{t-1})$.

Fig. 1: Inputs and outputs of a probabilistic motion model.

Different definitions for the $g(\cdot)$ function lead to different probabilistic motion models, like the velocity motion model explored here.

3.2.1 The model

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).

This motion model is characterized by the use of two velocities to control the robot's movement: **linear velocity** v and **angular velocity** w. Therefore, during the following sections, the movement commands will be of the form:

$$u_t = \begin{bmatrix} v_t \\ w_t \end{bmatrix}, \quad u_t \sim N(\bar{u}, \Sigma_{u_t})$$

The velocity motion model defines the function $g(\cdot)$ as:

$$g(x_{t-1}, u_t) = x_{t-1} \oplus \Delta x_t, \quad x_{t-1} \sim N(\bar{x}_{t-1}, \Sigma_{x_{t-1}})$$

being $\Delta_{x_t} = [\Delta_{x_t}, \Delta_{y_t}, \Delta_{\theta_t}]$ (assuming w and v constant):

- $\Delta x_t = \frac{v}{w}\sin(w\Delta t)$
- $\Delta y_t = \frac{v}{w}[1 - \cos(w\Delta t)]$
- $\Delta\theta_t = w\Delta t$

Note that $g(x_{t-1}, u_t) = x_{t-1} \oplus \Delta x_t$ **is not a linear operation!**

In this way, this motion model is characterized by the following equations, depending on the value of the angular velocity w (note that a division by zero would appear in the first case with $w = 0$):

- If $w \neq 0$:

$$\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{bmatrix} + \begin{bmatrix} -R\sin\theta_{t-1} + R\sin(\theta_{t-1} + \Delta\theta) \\ R\cos\theta_{t-1} - R\cos(\theta_{t-1} + \Delta\theta) \\ \Delta\theta \end{bmatrix}$$

- If $w = 0$:

$$\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{bmatrix} + v \cdot \Delta t \begin{bmatrix} \cos\theta_{t-1} \\ \sin\theta_{t-1} \\ 0 \end{bmatrix}$$

with:

- $v = w \cdot R$ *(R is also called the curvature radius)*
- $\Delta\theta = w \cdot \Delta t$

In [1]:
```
%matplotlib widget

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


import sys
sys.path.append("..")
from utils.DrawRobot import DrawRobot
from utils.PlotEllipse import PlotEllipse
```

ASSIGNMENT 1: The model in action

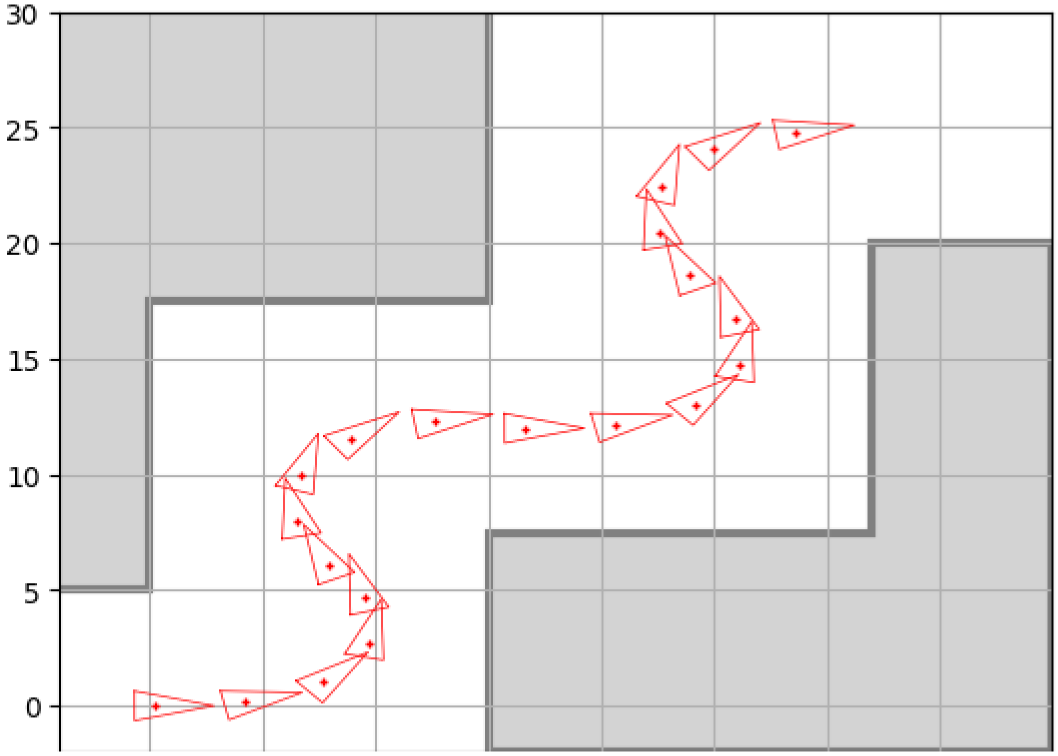
Modify the following `next_pose()` function, used in the `VelocityRobot` class below, which computes the next pose x_t of a robot given:

- its previous pose x_{t-1},
- the velocity movement command $u = [v, w]^T$, and
- a lapse of time Δt.

Concretly you have to complete the if-else statement that takes into account when the robot moves in an straight line so $w = 0$. *Note: you don't have to modify the* `None` *in the function header nor in the* `if cov is not None:` *condition.*

Remark that at this point **we are not taking into account uncertainty in the system**: neither from the initial pose ($\Sigma_{x_{t-1}}$) nor the movement (v, w) (Σ_{u_t}).

Example



In [15]:
```
def next_pose(x, u, dt, cov=None):
    ''' This function takes pose x and transform it according to the
        applying the differential drive model.

        Args:
            x: current pose
            u: differential command as a vector [v, w]'
            dt: Time interval in which the movement occurs
            cov: covariance of our movement. If not None, then add ga
    '''
    if cov is not None:
        u += np.sqrt(cov) @ random.randn(2, 1)
        #u = np.random.multivariate_normal(u.flatten(),cov)

    if u[1] == 0: #linear motion w=0
        next_x = np.vstack([x[0] + u[0] * dt * np.cos(x[2]),
                           x[1] + u[0] * dt * np.sin(x[2]),
                           x[2]])
    else: #Non-linear motion w=!0
        R = u[0]/u[1] #v/w=r is the curvature radius
        next_x = np.vstack([x[0] - R * np.sin(x[2]) + R * np.sin(x[2]
                           x[1] + R * np.cos(x[2]) - R * np.cos(x[2] + u
                           x[2] + u[1] * dt])

    return next_x
```

In [16]:
```
class VelocityRobot(object):
    """ Mobile robot implementation that uses velocity commands.

        Attr:
            pose: expected pose of the robot in the real world (witho
            dt: Duration of each step in seconds
    """
    def __init__(self, mean, dt):
        self.pose = mean
        self.dt = dt

    def step(self, u):
        self.pose = next_pose(self.pose, u, self.dt)

    def draw(self, fig, ax):
        DrawRobot(fig, ax, self.pose)
```

Test the movement of your robot using the demo below.

In [17]:
```
def main(robot, nSteps):

    v = 1 # Linear Velocity
    l = 0.5 #Half the width of the robot

    # MATPLOTLIB
    fig, ax = plt.subplots()
    plt.ion()
    fig.canvas.draw()
    plt.xlim((-2, 20))
    plt.ylim((-2, 30))
    plt.fill([7.5, 7.5, 16, 16, 20, 20],[-2, 7.5, 7.5, 20, 20, -2],
             facecolor='lightgray', edgecolor='gray', linewidth=3)
    plt.fill([-3, 0, 0, 7.5, 7.5, -3],[5, 5, 17.5, 17.5, 32, 32],
             facecolor='lightgray', edgecolor='gray', linewidth=3)

    plt.grid()

    # MAIN LOOP
    for k in range(1, nSteps + 1):
        #control is a wiggle with constant linear velocity
        u = np.vstack((v, np.pi / 10 * np.sin(4 * np.pi * k/nSteps)))

        robot.step(u)

        #draw occasionally
        if (k-1)%20 == 0:
            robot.draw(fig, ax)
            clear_output(wait=True)
            display(fig)
            time.sleep(0.1)

    plt.close()
```

In [18]:
```
# RUN
dT = 0.1 # time steps size
pose = np.vstack([0., 0., 0.])

robot = VelocityRobot(pose, dT)
main(robot, nSteps=400)
```

3.2.2 Propagating uncertainty

In the previous section we introduced how to compute the robot pose x at time instant t by applying a control action u_t. However, as we know, this process has different sources of uncertainty that need to be modeled someway.

To deal with this we will consider two Gaussian distributions:

- the **robot pose** modeled as $x_t \sim (\overline{x}_t, \Sigma_{x_t})$ at time t. Similarly, for the **previous pose** at $t-1$ we have $x_{t-1} \sim (\overline{x}_{t-1}, \Sigma_{x_{t-1}})$,
- and the **movement command** as $u_t \sim (\overline{u}_t, \Sigma_{u_t})$, being applied during an interval of time Δt.

In this way, after a motion command we can retrieve the probability distribution x_t modeling the new robot pose as:

- **Mean:**

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

- **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, and the covariance matrix of this movement (Σ_{u_t}) is defined as seen below. Typically, it is constant during robot motion:

OPTIONAL

Write a Markdown cell containing the Jacobians ecuations aforementioned.

END OF OPTIONAL PART

ASSIGNMENT 2: Adding uncertainty

Now we will include uncertainty to the previous assignment, changing the behavior of the robot class `VelocityRobot()` you have implemented.

In contrast to the noisy robot `NoisyRobot()` in notebook 3.1, we will use the equations of the velocity motion model and their respective Jacobians to keep track of how confident we are of the robot's pose (i.e. the robot's pose x_t now is also a gaussian distribution).

Consider the following:

- the expected robot pose $\overline{x}_t$ is stored in `self.pose` .
- the covariance matrix of the robot pose Σ_{x_t} is named `P_t` in the code,
- the covariance matrix of the robot motion Σ_{u_t} is `Q` , and
- the jacobians of our motion model $\partial g/\partial x_{t-1}$ and $\partial g/\partial u_t$ are `JacF_x` and `JacF_u` .

First Complete the following code calculating the covariance matrix Σ_{x_t} (`P_t`). That is, you have to:

- Implement the jacobians `JacF_x` and `JacF_u` , which depend on the angular velocity w, and
- Compute the covariance matrix `P_t` using such jacobians, the current covariance of the pose `P` , and the covariance of the motion `Q` .

In [20]:
```
def next_covariance(x, P, Q, u, dt):
    ''' Compute the covariance of a robot following the velocity mot:

        Args:
            x: current pose (before movement)
            u: differential command as a vector [v, w]''
            dt: Time interval in which the movement occurs
            P: current covariance of the pose
            Q: covariance of our movement.
    '''
    # Aliases
    v = u[0, 0]
    w = u[1, 0]

    sx, cx = np.sin(x[2, 0]), np.cos(x[2, 0]) #sin and cos for the pı
    si, ci = np.sin(u[1, 0]*dt), np.cos(u[1, 0]*dt) #sin and cos for
    R = u[0, 0]/u[1, 0] #v/w Curvature radius

    if u[1, 0] == 0:  #linear motion w=0 --> R = infinite
        #TODO JACOBIAN HERE
        JacF_x = np.array([
            [1,0,-v * dt * np.sin(x[2,0])],
            [0,1,v * dt * np.cos(x[2,0])],
            [0,0,1]
        ])
        JacF_u = np.array([
            [dt * np.cos(x[2,0]), 0],
            [dt * np.sin(x[2,0]), 0],
            [0,0]
        ])
    else: #Non-linear motion w=!0
        # TODO JACOBIAN HERE
        JacF_x = np.array([
            [1,0, R * (-sx * si - cx * (1-ci))],
            [0,1, R * (cx * si - sx * (1-ci))],
            [0,0,1]
        ])

        JacF_u = (
            np.array([
                [cx * si - sx * (1-ci), R * (cx*ci - sx*si)],
                [sx * si + cx * (1-ci), R * (sx*ci - cx*si)],
                [0,1]
            ])@
            np.array([
                [1/w, -v/w**2],
                [0, dt]
            ])
        )
    #prediction steps
    Pt = ( JacF_x @ P @ JacF_x.transpose() ) + ( JacF_u @ Q @ JacF_u.

    return Pt
```

Then, complete the methods:

- `step()` to get the true robot pose (ground-truth) using the Q matrix (recall the `next_pose()` function you defined before and its fourth input argument),

- and the `draw()` one to plot an ellipse representing the uncertainty about the robot pose centered at the expected robot pose (`self.pose`) as well as marks representing the ground truth poses.

Example

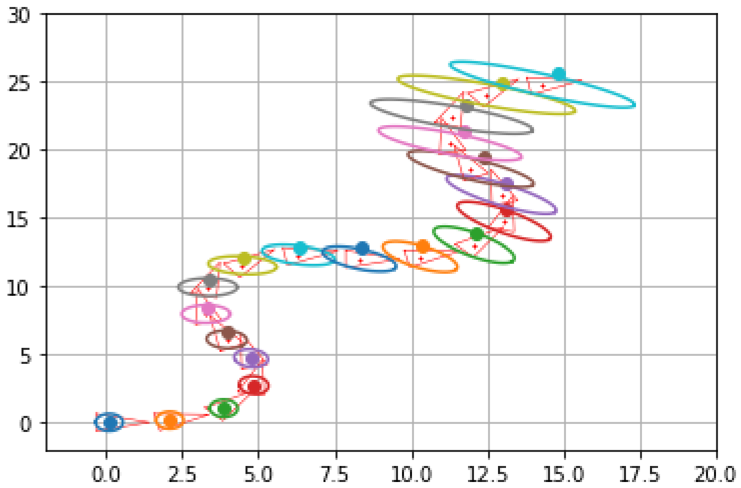
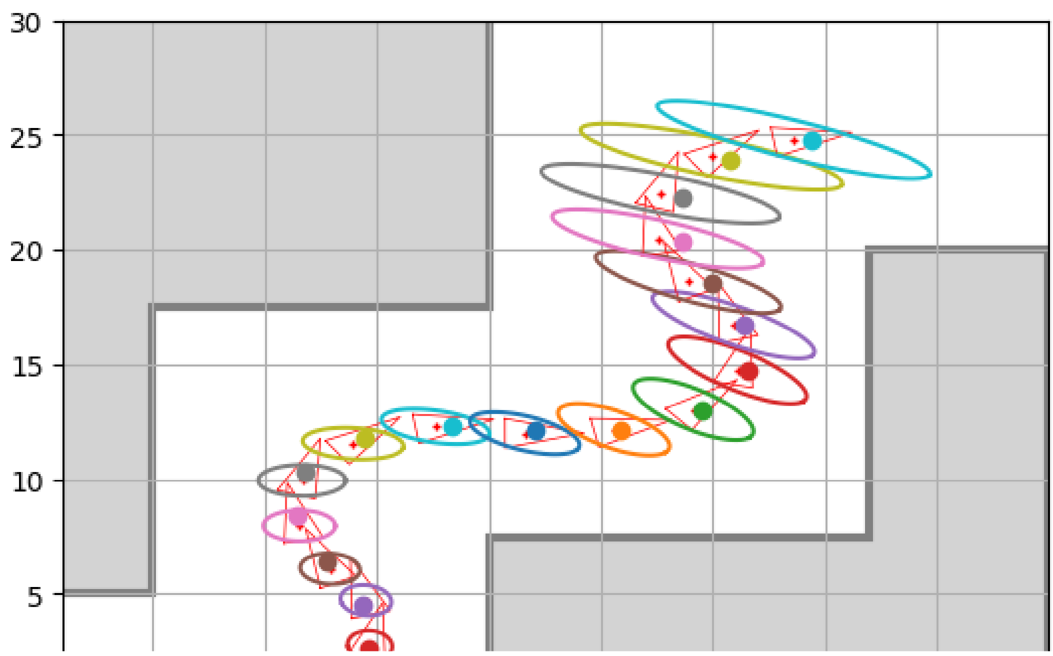


Fig. 3: Movement of a robot using velocity commands.
Representing the expected pose (in red), the true pose (as dots)

In [21]:
```
class NoisyVelocityRobot(VelocityRobot):
    """ Mobile robot implementation that uses velocity commands.

        Attr:
            [...]: Inherited from VelocityRobot
            true_pose: expected pose of the robot in the real world (
            cov_pose: Covariance of the pose at each step
            cov_move: Covariance of each movement. It is a constant

    """

    def __init__(self, mean, cov_pose, cov_move, dt):
        super().__init__(mean, dt)
        self.true_pose = mean
        self.cov_pose = cov_pose
        self.cov_move = cov_move

    def step(self, u):
        self.cov_pose = next_covariance(self.pose, self.cov_pose, sel

        super().step(u)
        self.true_pose = next_pose(self.true_pose, u, self.dt, cov=se

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

Now, **try your implementation!**

In [23]:
```
# RUN
dT = 0.1 # time steps size

SigmaV = 0.2 #Standard deviation of the linear velocity.
SigmaW = 0.1 #Standard deviation of the angular velocity
nSteps = 400 #Number of motions

P = np.diag([0.2, 0.4, 0.]) #pose covariance matrix 3x3
Q = np.diag([SigmaV**2, SigmaW**2]) #motion covariance matrix 2x2

robot = NoisyVelocityRobot(np.vstack([0., 0., 0.]), P, Q, dT)
main(robot, nSteps=nSteps)
```



Thinking about it (1)

Now that you have some experience with robot motion and the velocity motion model, **answer the following questions**:

- Why do we need to consider two different cases when applying the $g(\cdot)$ function, that is, calculating the new robot pose?

  Dependiendo de si el valor de la velocidad angular es nulo o no, hemos podido comprobar que tenemos dos casos. Esto es debido a que no podemos considerar de igual manera (aplicar las mismas ecuaciones) los cambios de pose que incluyen una rotación de los que no.

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

  u_t es un vector de dos componentes: La velocidad lineal (v) y la velocidad angular (w).

- Why do we need to use Jacobians to propagate the uncertainty about the robot pose x_t?

  Empleamos los jacobianos porque la operación de composición de poses es no lineal, por lo que tenemos que aproximar para hallar en un tiempo razonable la nueva matriz de covarianzas.

- 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.

De manera similar al cuaderno anterior en el que introducimos ruido, a medida que aumentamos los valores de la matriz, la incertidumbre representada en las elipses es mayor.

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:

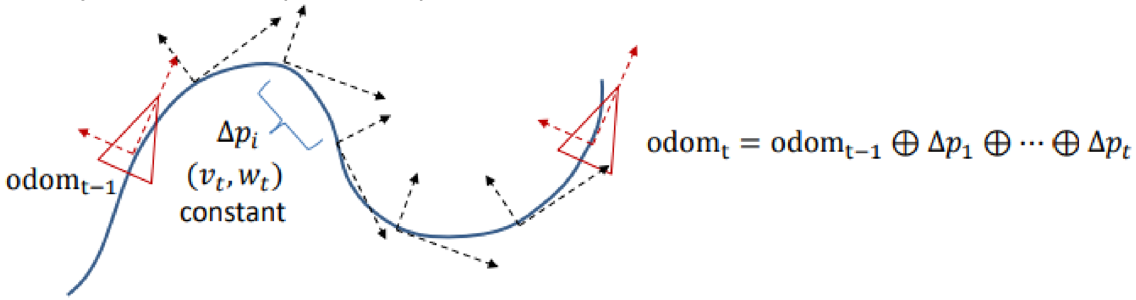


Fig. 2: Example of composition of poses based on odometry.

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 this way, the utilization of the odometry motion model is more suitable to keep track and estimate the robot pose in contrast to the *velocity model*: it is more accurate, but

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:

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

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

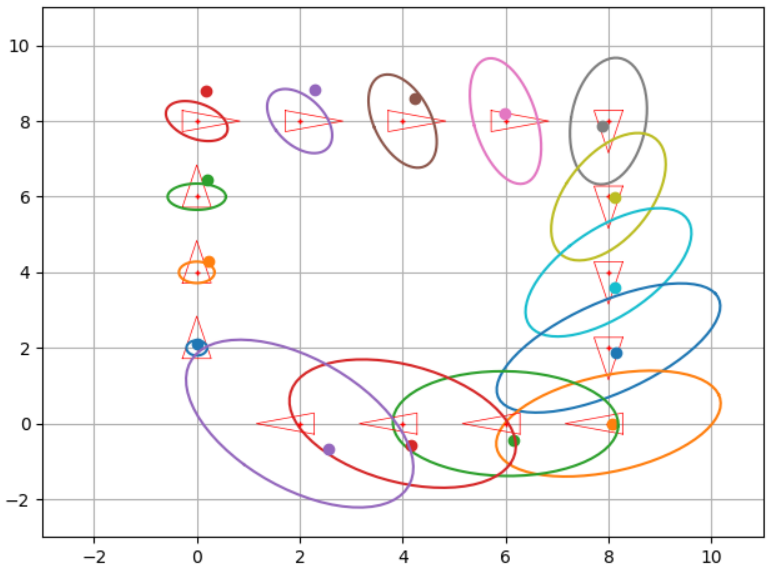


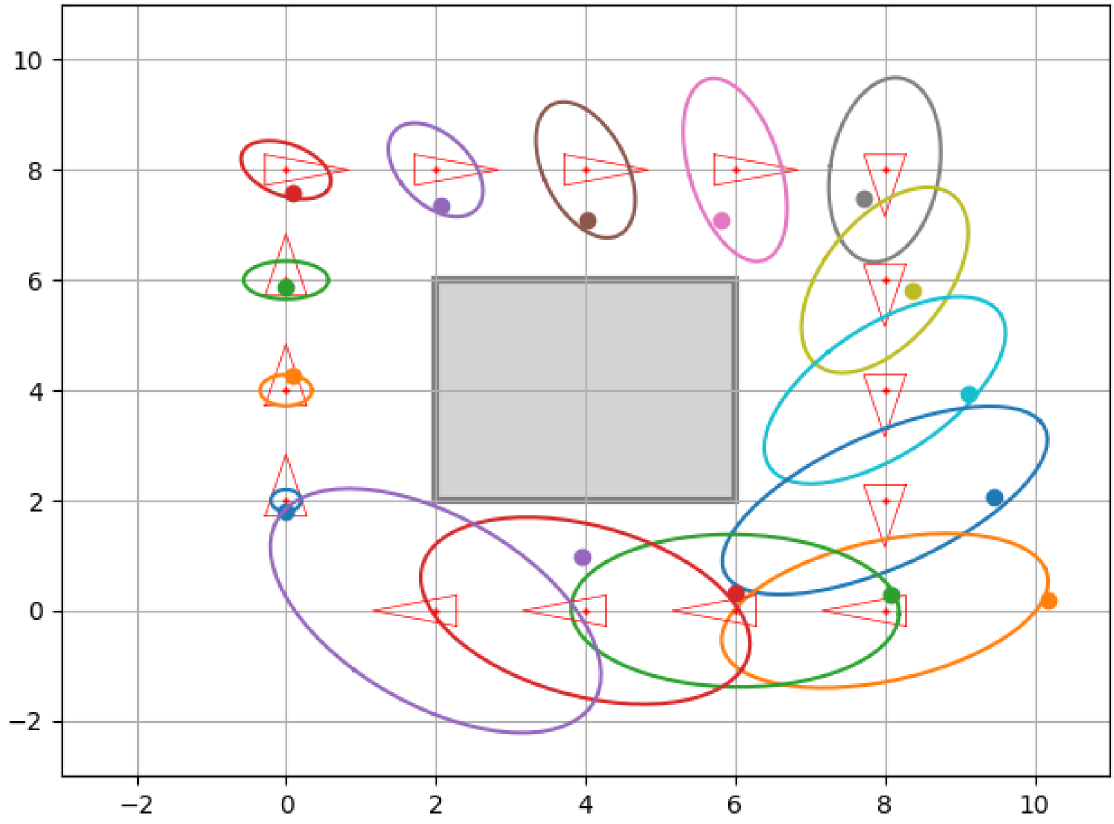
Fig. 2: Movement of a robot using odometry commands. Representing the expected pose (in red), the true pose (as dots) and the confidence ellipse.

In [2]:
```
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(mean=u.flat
        self.true_pose = tcomp(self.true_pose, noisy_u)

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

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

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

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', edgecol
    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 [4]:
```
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?

  Anteriormente vimos que la función g debía ser considerada en función de si la velocidad angular del incremento de pose era nulo o no, pero en odometría estamos comprobando como no hay que hacer dicha distinción, sino que puede ser considerada una única expresión descrita en los apuntes de este mismo cuaderno. Observando detenimadente la expresión podemos ver el motivo, ya que esta única expresión no contiene el termino w (velocidad angular) por lo que no tiene sentido la distinción.

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

  u_t tiene en esta ocasión tres parámetros: Δx_t (incremento en el eje X), Δy_t (incremento en el eje Y), $\Delta \theta_t$ (variación del ángulo)

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

  Gracias a ellos somos capaces de aproximar la covarianza de la nueva pose a partir de la pose anterior y el vector u (movimiento realizado).

- 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.

  Como es de esperar, a medida que aumentamos los valores la incertidumbre que tiene el robot sobre su posición es mayor y esto se refleja en la elipse que la representa (que es cada vez mayor).

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.

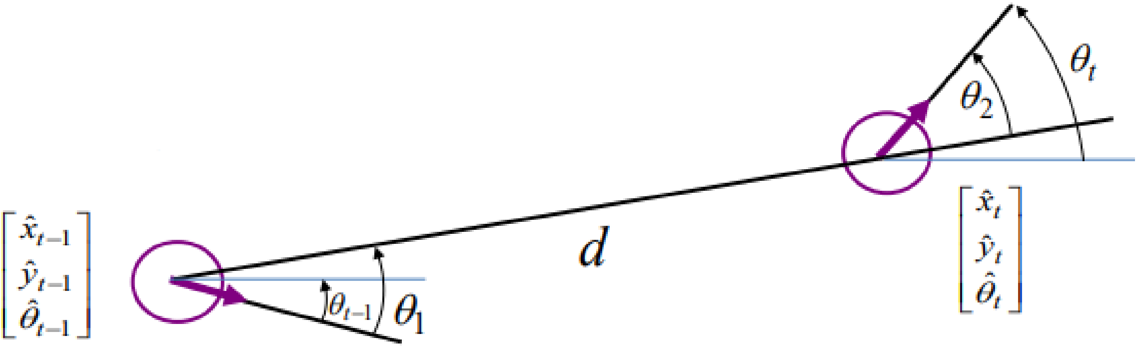


Fig. 3: Movement of a robot using odometry commands in sampling form.

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 [5]:
```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(np.power(diff[1],2) + np.power(diff[0],2))
    theta2 = diff[2] - theta1
    return np.vstack((theta1, d, theta2))
```

Try such function with the code cell below:

In [6]:
```python
generate_move(np.vstack([0., 0., 0.]), np.vstack([1., 1., np.pi/2]))
```

Out[6]:
```
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

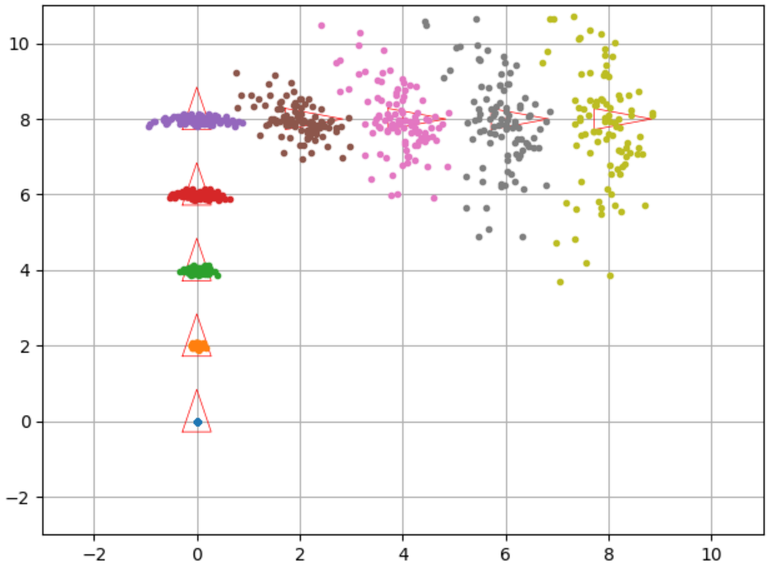


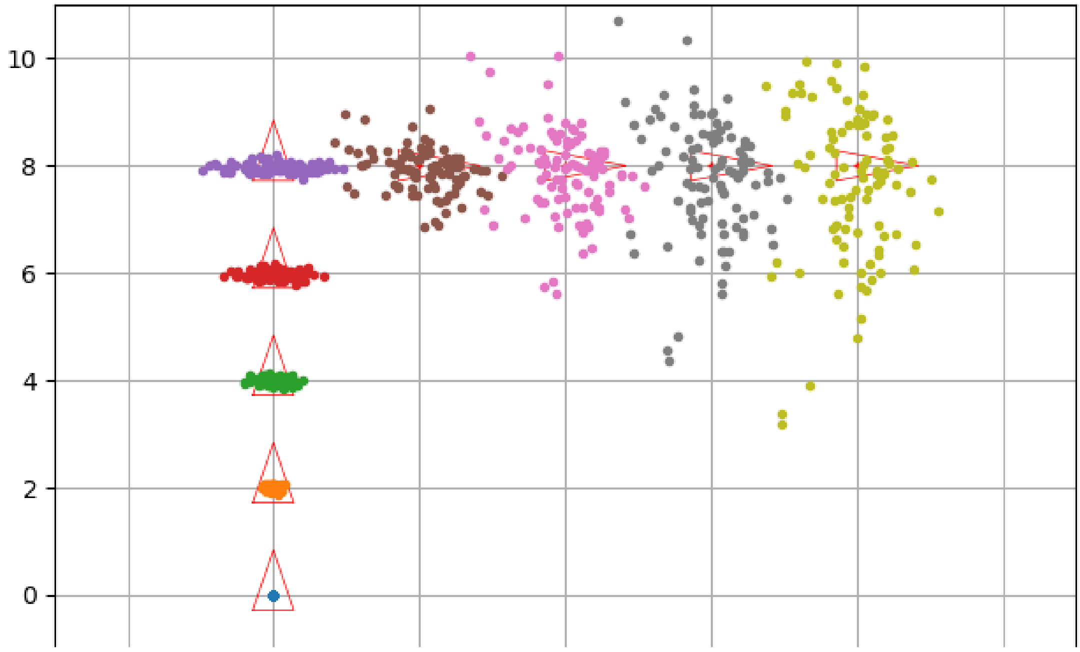
Fig. 1: Movement of a robot using odometry commands in sampling form. Representing the expected pose (in red) and the samples (as clouds of dots)

In [7]:
```
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] + u[0]
        self.pose[0] += u[1] * np.cos(ang)
        self.pose[1] += u[1] * np.sin(ang)
        self.pose[2] = ang + u[2]

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

        u2 = u**2

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

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

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

    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 [8]:
```
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 [9]:
```
# 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)?

  De manera similar a los anteriores cuadernos, a medida que aumentamos su valor (en el orden de las decenas de unidad) vemos como la dispersión en la dimensión correspondiente aumenta.

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

  Nos encontramos ante un total de tres parámetros: θ_1 (ángulo de giro para el punto), d (movimiento para llegar al destino) y θ_2 (ángulo de giro para llegar al destino)

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

  Se genera un efecto aspersor, con forma de arco.