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.

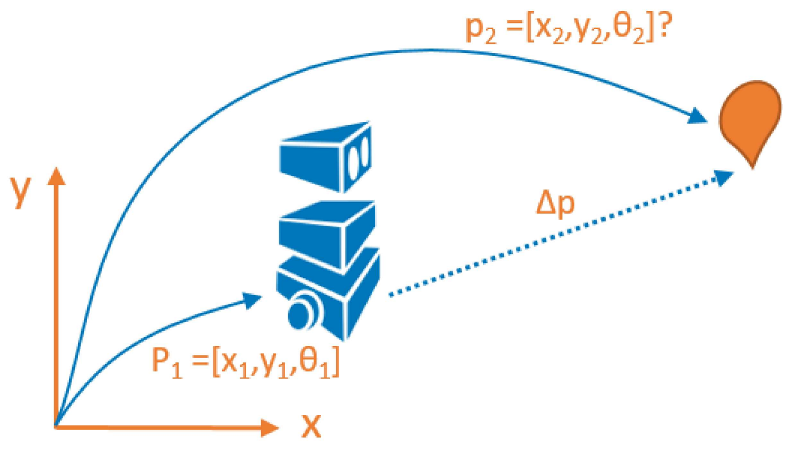


Fig. 1: Example of an initial 2D robot pose (p_1) and its resultant pose (p_2) after completing a motion (Δp).

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.

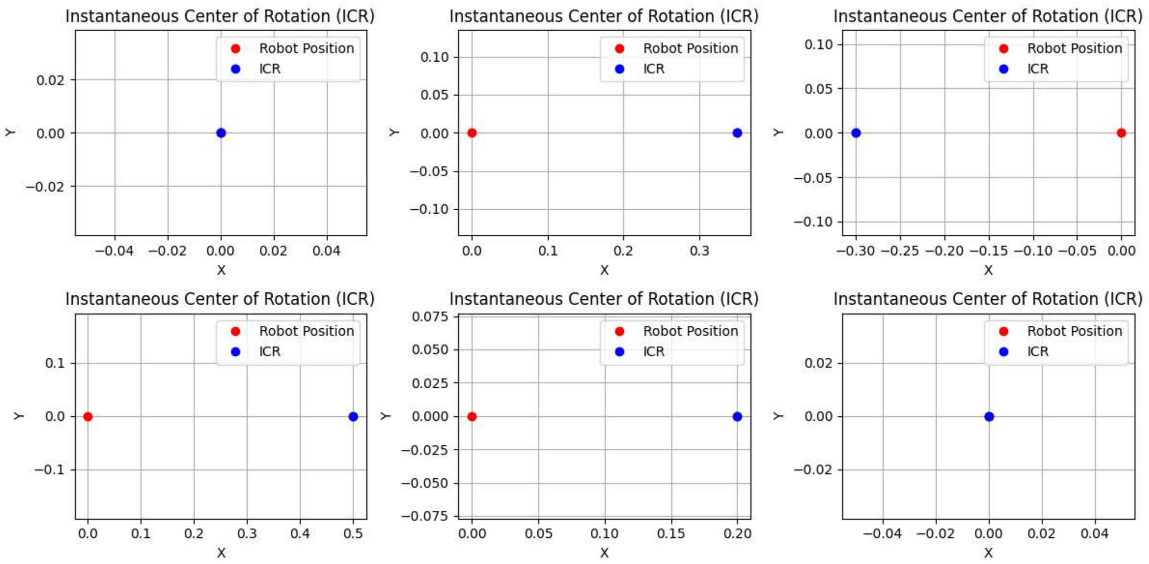
In [2]:
```
%matplotlib widget

# 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.

In [3]:
```
# R = Distance between wheels' axles (half the distance)
# left_wheel_velocity = Linear velocity of the left wheel
# right_wheel_velocity = Linear velocity of the right wheel (opposite direct
# ICR_x = R / 2 * (V_left + V_right)

def calculate_ICR(R, left_wheel_velocity, right_wheel_velocity): # Calculate
        ICR_x = R / 2 * (left_wheel_velocity + right_wheel_velocity)
        return ICR_x

def plot_ICR(R, left_wheel_velocity, right_wheel_velocity):

    ICR_x = calculate_ICR(R, left_wheel_velocity, right_wheel_velocity)

    plt.plot([0, ICR_x], [0, 0], 'ro', label="Robot Position") # Plot the ro
    plt.plot(ICR_x, 0, 'bo', label="ICR") # Plot the ICR

    plt.xlabel("X")
    plt.ylabel("Y")
    plt.title("Instantaneous Center of Rotation (ICR)")
    plt.legend()
    plt.grid(True)
    plt.axis('equal')

# Ejemplos
plt.figure(figsize=(12, 6))
plt.subplot(231)
plot_ICR(0.5, 1.0, -1.0)
plt.subplot(232)
plot_ICR(0.7, 2, -1)
plt.subplot(233)
plot_ICR(0.6, 1, -2)
plt.subplot(234)
plot_ICR(1.0, 1.5, -0.5)
plt.subplot(235)
plot_ICR(0.4, 1.5, -0.5)
plt.subplot(236)
plot_ICR(0.8, 1.5, -1.5)

plt.tight_layout()
plt.show()
```

Figure



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:

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

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

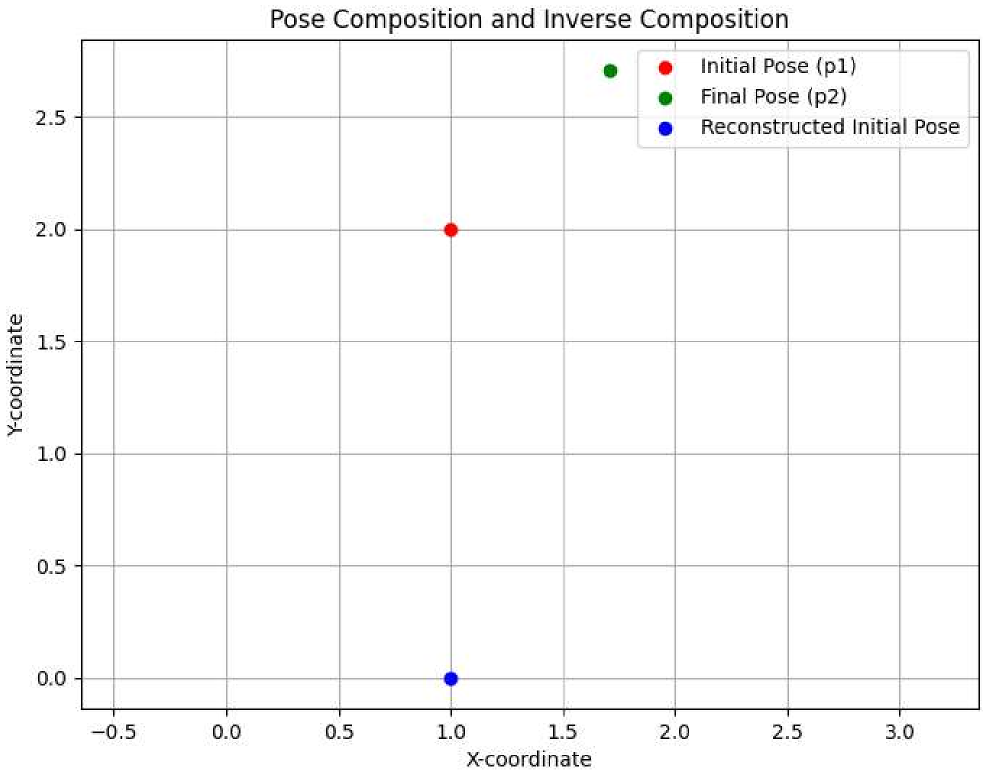
The differential Δp, although we are using it as control in this exercise, normally is calculated given the robot's locomotion or sensed by the wheel encoders.

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.

In [4]:
```
# Function to compute the composition of two poses. (Formulas above).
def compose_poses(p1, dp):
    x1, y1, theta1 = p1
    delta_x, delta_y, delta_theta = dp

    x = x1 + delta_x * np.cos(theta1) - delta_y * np.sin(theta1)
    y = y1 + delta_x * np.sin(theta1) + delta_y * np.cos(theta1)
    theta = theta1 + delta_theta

    return np.array([x, y, theta])

# Function to compute the inverse composition of two poses. (Class notebook)
def inverse_compose_poses(p1, p2):
    x1, y1, theta1 = p1
    x2, y2, theta2 = p2

    delta_x = (x2 - x1) * np.cos(-theta1) - (y2 - y1) * np.sin(-theta1)
    delta_y = (x2 - x1) * np.sin(-theta1) + (y2 - y1) * np.cos(-theta1)
    delta_theta = theta2 - theta1

    return np.array([delta_x, delta_y, delta_theta])

# Example usage and plotting
p1 = np.array([1.0, 2.0, np.pi/4])  # Initial pose (1, 2) with angle pi/4
dp = np.array([1.0, 0.0, np.pi/6])  # Pose differential (1,0) with angle pi/

# Calculate the final pose using composition
p2 = compose_poses(p1, dp)

# Calculate the original pose (p1) using inverse composition
reconstructed_p1 = inverse_compose_poses(p1, p2)

# Plot the initial pose, final pose, and reconstructed initial pose
plt.figure(figsize=(8, 6))
plt.plot(p1[0], p1[1], 'ro', label='Initial Pose (p1)')
plt.plot(p2[0], p2[1], 'go', label='Final Pose (p2)')
plt.plot(reconstructed_p1[0], reconstructed_p1[1], 'bo', label='Reconstructe
plt.xlabel('X-coordinate')
plt.ylabel('Y-coordinate')
plt.title('Pose Composition and Inverse Composition')
plt.legend()
plt.grid(True)
plt.axis('equal')

plt.show()
```

Figure



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

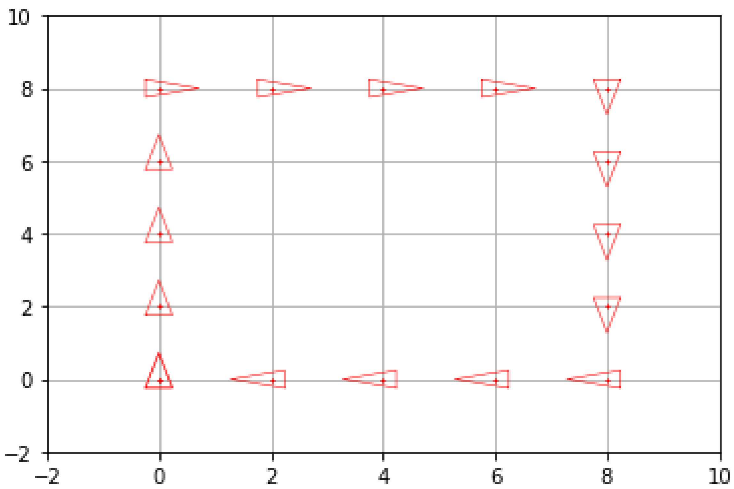


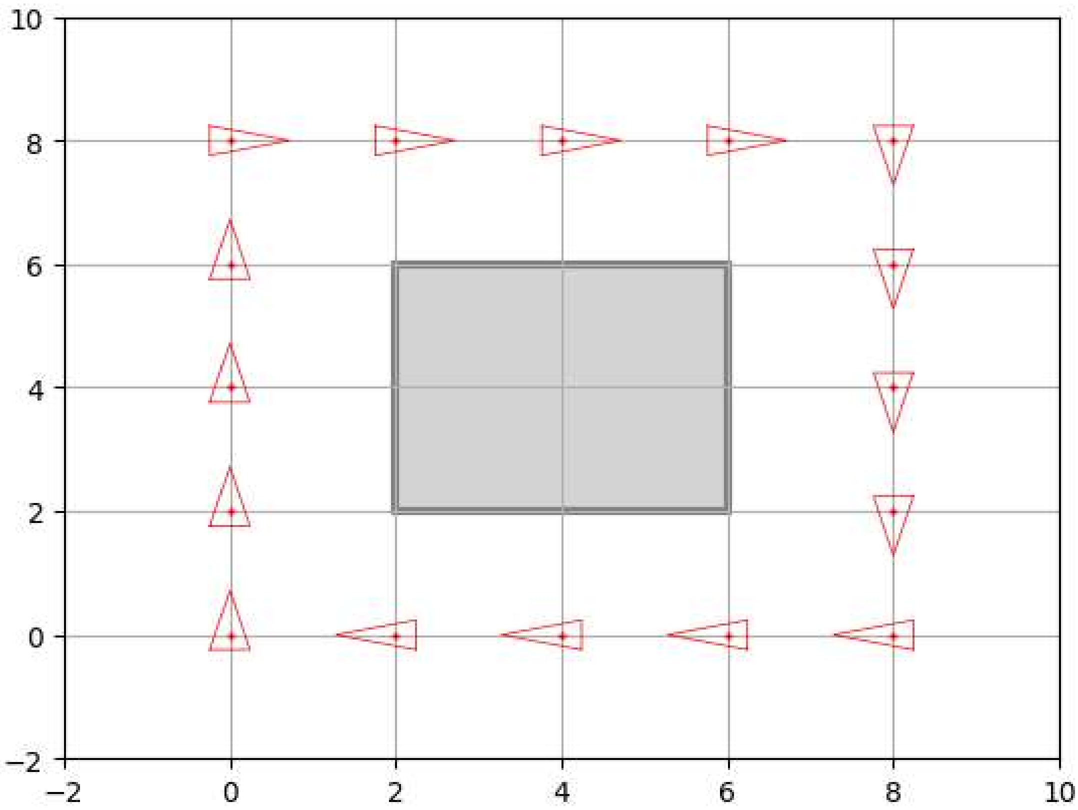
Fig. 2: Route of our robot.

In [5]:
```
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 [6]:
```
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', edgecolor='gra

    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 to tur
        # and accordingly set the third component (rotation) of the motion c
        if step % turning == 0:
            u[2] = angle_inc  # Turn right --> -np.pi/2
        else:
            u[2] = 0.0  # Move forward

        # 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 [7]:
```
# 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.html) *(https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.multivariate_normal.html)*
  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

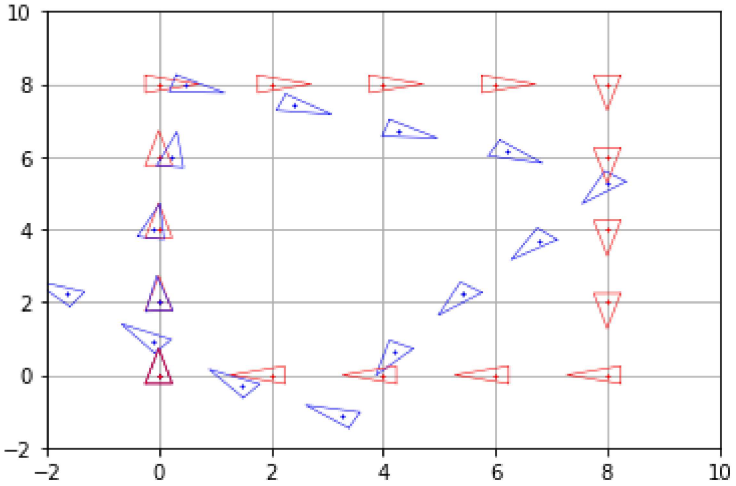
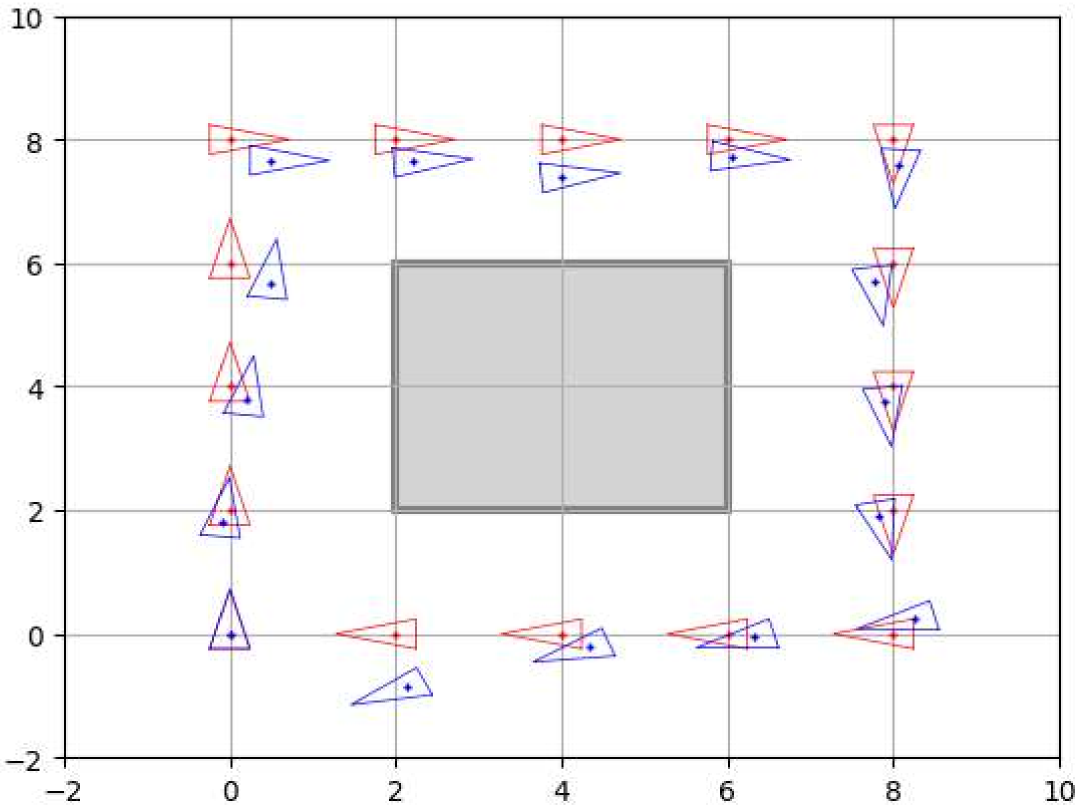


Fig. 3: Movement of our robot using pose compositions. Containing the expected poses (in red) and the true pose affected by noise (in blue)

In [8]:
```
class NoisyRobot(Robot):
    """Mobile robot implementation. It's motion has a set ammount of noise.

        Attr:
            pose: Inherited from Robot
            true_pose: Real robot pose, which has been affected by some ammo
            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 noise)
            Generate a noisy increment based on step_increment and self.cova
            Then this noisy increment is applied to self.true_pose
        """
        super().step(step_increment)
        true_step = stats.multivariate_normal.rvs(step_increment.flatten(),
        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 [9]:
```
# RUN
initial_pose = np.vstack([0., 0., np.pi/2])
cov = np.diag([0.04, 0.04, 0.01])

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?

  The expected (red) and true (blue) poses are different because of the introduction of noise in the robot's motion. When we apply the increment, we add noise to it based on the covariance matrix. This noise introduces uncertainty in the robot's actual position, causing it to deviate from the expected ideal position.

- In which scenario could they be the same?

  The expected and true poses could be the same in a scenario where there is no uncertainty or noise in the robot's motion. If the covariance matrix is set to all zeros, it means that there is no noise or uncertainty.

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

  Larger (Smaller) values in the covariance matrix represent higher (lower) uncertainty, which means that the robot's motion will be more (less) affected by noise, leading to larger (smaller) deviations between the expected and true poses. The values in the covariance matrix are distributed on the diagonal, with each value corresponding to a specific dimension of the robot's pose: - The value at (1,1) represents the error (covariance) in the X-axis - The value at (2,2) represents the error (covariance) in the Y-axis - The value at (3,3) represents the error (covariance) in the angle theta