# 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,
- temporal response of motors,
- 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.

After executing a motion command, the robot would end up in a different position/orientation from the initial one. This particular chapter explores the concept of *robot's pose* used to represent these positions/orientations, 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.

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.

## Notebook context: move that robot!

The figure below shows a Giraff robot, equipped with a rig of RGB-D sensors and a 2D laser scanners. The robot is gathering information from said sensors to collect a dataset. Datasets are useful to train and test new techniques for navigation, perception, etc.

However, if the robot remains static, the dataset will only contain information about the part of the room that it is currently inspecting, so, we need to move it!

Your task in this notebook will be to command the robot to move through the environment and calculate its new position after executing a motion command. Let's go!

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.

He desarrollado un código que permite conocer el Centro Instántaneo de Rotación (ICR) de un robot con dos ruedas que únicamente modifican su velocidad. Para ello, se da como parámetro la distancia entre las ruedas y la velocidad de cada una de ellas, de forma que se calcula el ICR usando la fórmula:

$$ICR_x = 0$$

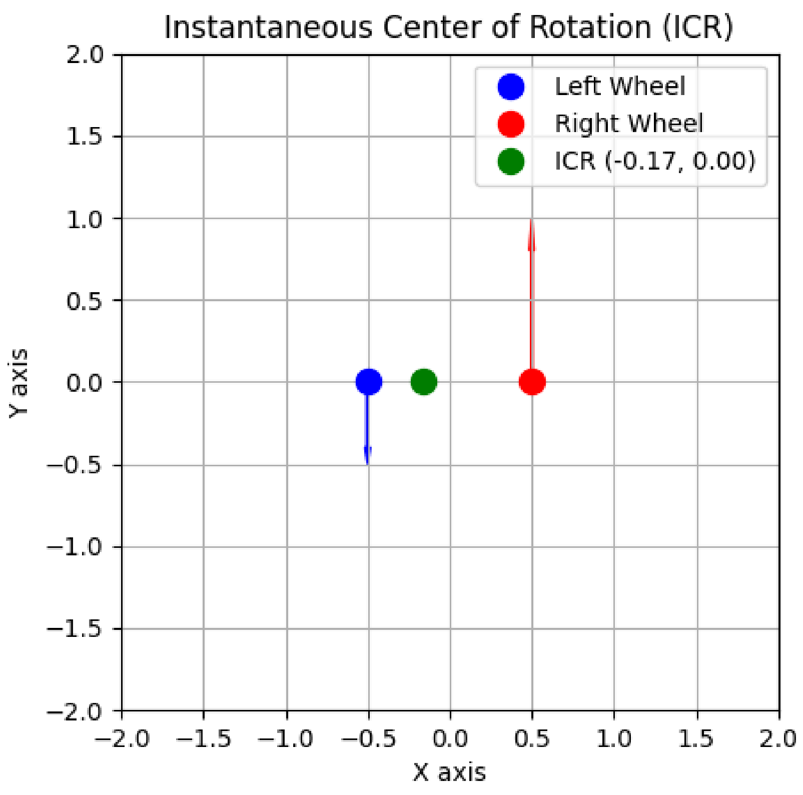
$$ICR_y = (distancia_{ruedas}/2) \cdot \frac{v_{derecha} + v_{izquierda}}{v_{derecha} - v_{izquierda}}$$

In [2]:
```
from matplotlib.patches import Arrow
```

In [3]:
```
def plot_icr(wheel_base, left_wheel_speed, right_wheel_speed):
    """
    Plots the Instantaneous Center of Rotation (ICR) for a differential drive ro

    Parameters:
    - wheel_base: Distance between the left and right wheels.
    - left_wheel_speed: Speed of the left wheel.
    - right_wheel_speed: Speed of the right wheel.
    """
    # Calculate the radius of curvature
    if left_wheel_speed == right_wheel_speed:
        radius = float('inf')  # Straight line motion
        print('Straight line motion, the ICR is at infinity.')
        return
    else:
        radius = (wheel_base / 2) * ((right_wheel_speed + left_wheel_speed) / (r
        if right_wheel_speed > left_wheel_speed:
            radius = -abs(radius)
        else:
            radius = abs(radius)
    icr_x = radius
    icr_y = 0

    # Plotting
    fig, ax = plt.subplots()
    ax.plot(-(wheel_base/2), 0, 'bo', markersize=10)  # Left wheel
    ax.plot(wheel_base/2, 0, 'ro', markersize=10)  # Right wheel
    ax.plot(icr_x, icr_y, 'go', markersize=10)  # ICR

    # Draw arrows for wheel speeds without adding them to the legend
    ax.add_patch(Arrow(-(wheel_base/2), 0, 0, left_wheel_speed, color='blue', wi
    ax.add_patch(Arrow(wheel_base/2, 0, 0, right_wheel_speed, color='red', width

    # Adjust axis limits to ensure ICR is visible
    axis_limit = max(abs(radius), wheel_base) + 1
    ax.set_xlim(-axis_limit, axis_limit)
    ax.set_ylim(-axis_limit, axis_limit)
    # Show only wheels and ICR in the legend
    ax.legend(['Left Wheel', 'Right Wheel', f'ICR ({icr_x:.2f}, {icr_y:.2f})'])

    ax.set_aspect('equal', 'box')
    ax.grid(True)
    plt.title('Instantaneous Center of Rotation (ICR)')
    plt.xlabel('X axis')
    plt.ylabel('Y axis')
    plt.show()
```

In [4]:
```
wheel_base = 1  # Distance between wheels
left_wheel_speed = -0.5 # Speed of the left wheel
right_wheel_speed = 1.0  # Speed of the right wheel

plot_icr(wheel_base, left_wheel_speed, right_wheel_speed)
```

Figure



## 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 $\Delta p$ (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 (see Fig.1):

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

$$p_2 = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} = p_1 \oplus \Delta p = \begin{bmatrix} x_1 + \Delta x \cos\theta_1 - \Delta y \sin\theta_1 \\ y_1 + \Delta x \sin\theta_1 + \Delta y \cos\theta_1 \\ \theta_1 + \Delta\theta \end{bmatrix} \tag{1}$$

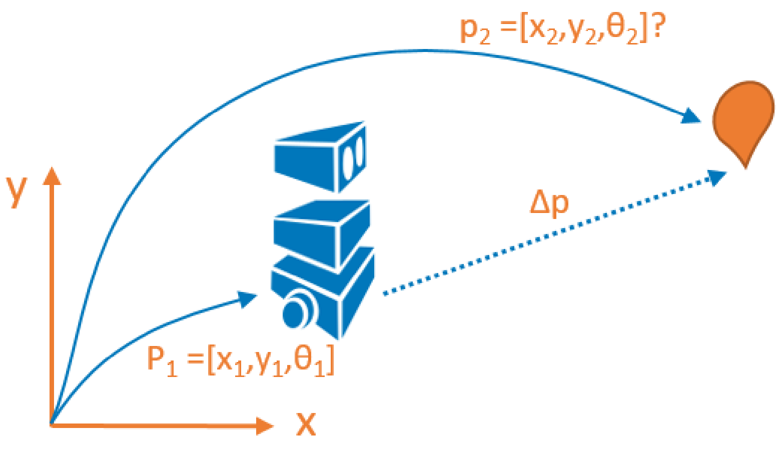


Fig. 1: Example of an initial 2D robot pose ($p_1$) and its resultant pose ($p_2$) after completing a motion ($\Delta p$).

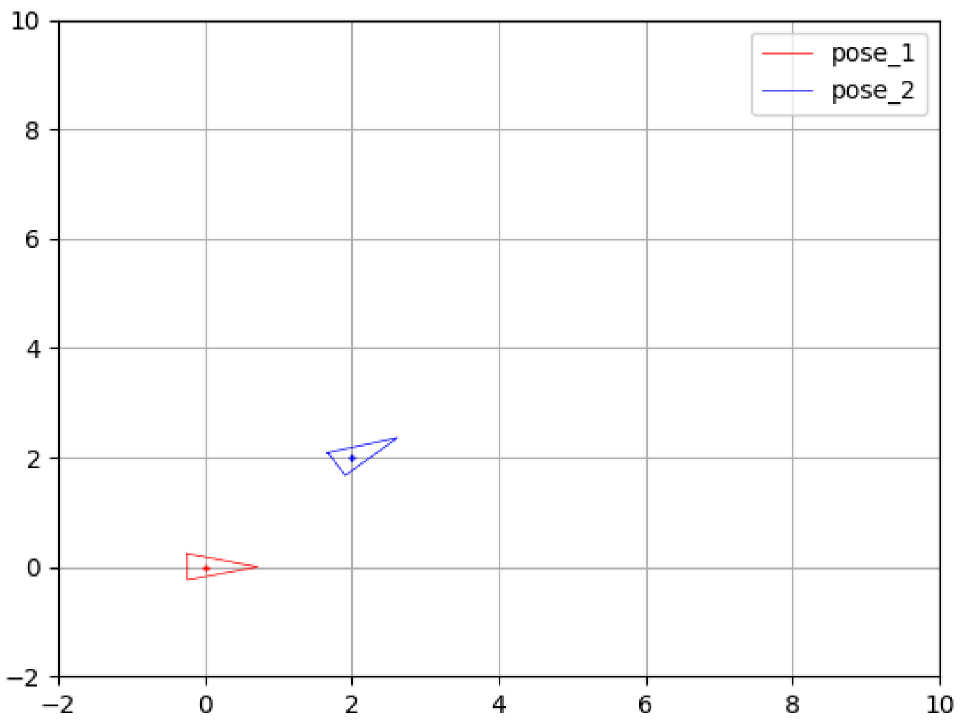
The differential $\Delta p$, although we are using it as control in this exercise, normally is calculated given the robot's locomotion or sensed by the wheel encoders.

You are provided with a function called `pose_2 = tcomp(pose_1,u)` that apply the composition of poses to pose `pose_1` and pose increment `u` and returns the new pose `pose_2`. Below you have a code cell to play with it.

In [5]:
```
# Pose increments' playground!

# You can modify pose and increment here to experiment
pose_1 = np.vstack([0, 0, 0])  # Initial pose
u = np.vstack([2,2, np.pi/6])  # Pose increment
pose_2 = tcomp(pose_1, u) # Pose after executing the motion

# NUMERICAL RESULTS
print(f"Initial pose: {pose_1}")
print(f"Pose increment: {u}")
print(f"New pose after applying tcomp: {pose_2}")

# VISUALIZATION
fig, ax = plt.subplots()
plt.grid('on')
plt.xlim((-2, 10))
plt.ylim((-2, 10))
h1 = DrawRobot(fig, ax, pose_1);
h2 = DrawRobot(fig, ax, pose_2, color='blue')
plt.legend([h1[0],h2[0]],['pose_1','pose_2']);
```

```
Initial pose: [[0]
 [0]
 [0]]
Pose increment: [[2.        ]
 [2.        ]
 [0.52359878]]
New pose after applying tcomp: [[2.        ]
 [2.        ]
 [0.52359878]]
```

Figure



## OPTIONAL

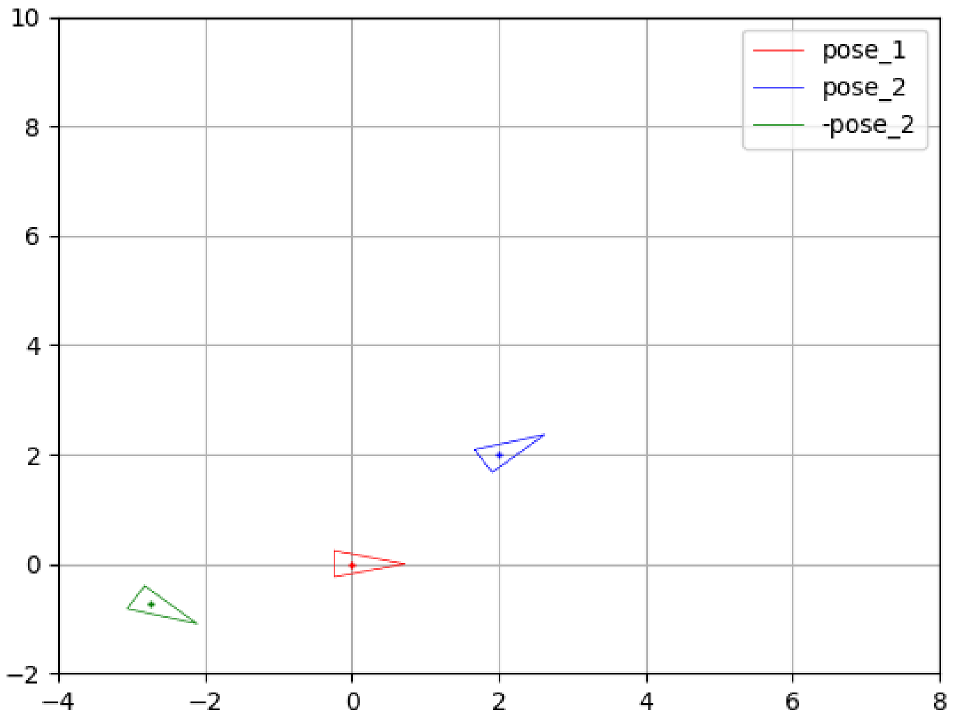
Implement your own methods to compute the composition of two poses, as well as the inverse composition. Include some examples of their utilization, also incorporating plots.

In [6]:
```python
def compose_poses(p1, p2):
    """
    Compose two poses p1 and p2.
    """
    x1, y1, theta1 = p1.flatten()
    x2, y2, theta2 = p2.flatten()

    x = x1 + x2 * np.cos(theta1) - y2 * np.sin(theta1)
    y = y1 + x2 * np.sin(theta1) + y2 * np.cos(theta1)
    theta = theta1 + theta2

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

def inverse_pose(p):
    """
    Compute the inverse of a pose.
    """
    x, y, theta = p.flatten()

    x_inv = -x * np.cos(theta) - y * np.sin(theta)
    y_inv = x * np.sin(theta) - y * np.cos(theta)
    theta_inv = -theta

    return np.vstack([x_inv, y_inv, theta_inv])
```

In [7]:
```
pose_1 = np.vstack([0, 0, 0])  # Initial pose
u = np.vstack([2,2, np.pi/6])  # Pose increment
pose_2 = compose_poses(pose_1, u) # Pose after executing the motion
pose_3 = inverse_pose(pose_2) # Inverse of a pose

# NUMERICAL RESULTS
print(f"Initial pose: {pose_1}")
print(f"Pose increment: {u}")
print(f"New pose after applying tcomp: {pose_2}")
print(f"Inverse pose of pose_2: {pose_3}")

# VISUALIZATION
fig, ax = plt.subplots()
plt.grid('on')
plt.xlim((-4, 8))
plt.ylim((-2, 10))
h1 = DrawRobot(fig, ax, pose_1);
h2 = DrawRobot(fig, ax, pose_2, color='blue')
h3 = DrawRobot(fig, ax, pose_3, color='green')
plt.legend([h1[0],h2[0],h3[0]],['pose_1','pose_2','-pose_2']);
```

```
Initial pose: [[0]
 [0]
 [0]]
Pose increment: [[2.        ]
 [2.        ]
 [0.52359878]]
New pose after applying tcomp: [[2.        ]
 [2.        ]
 [0.52359878]]
Inverse pose of pose_2: [[-2.73205081]
 [-0.73205081]
 [-0.52359878]]
```

Figure



## *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**

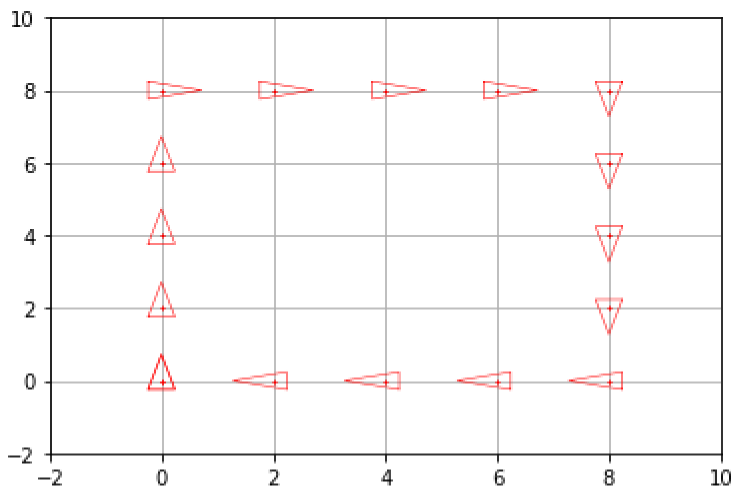


Fig. 2: Route of our robot.

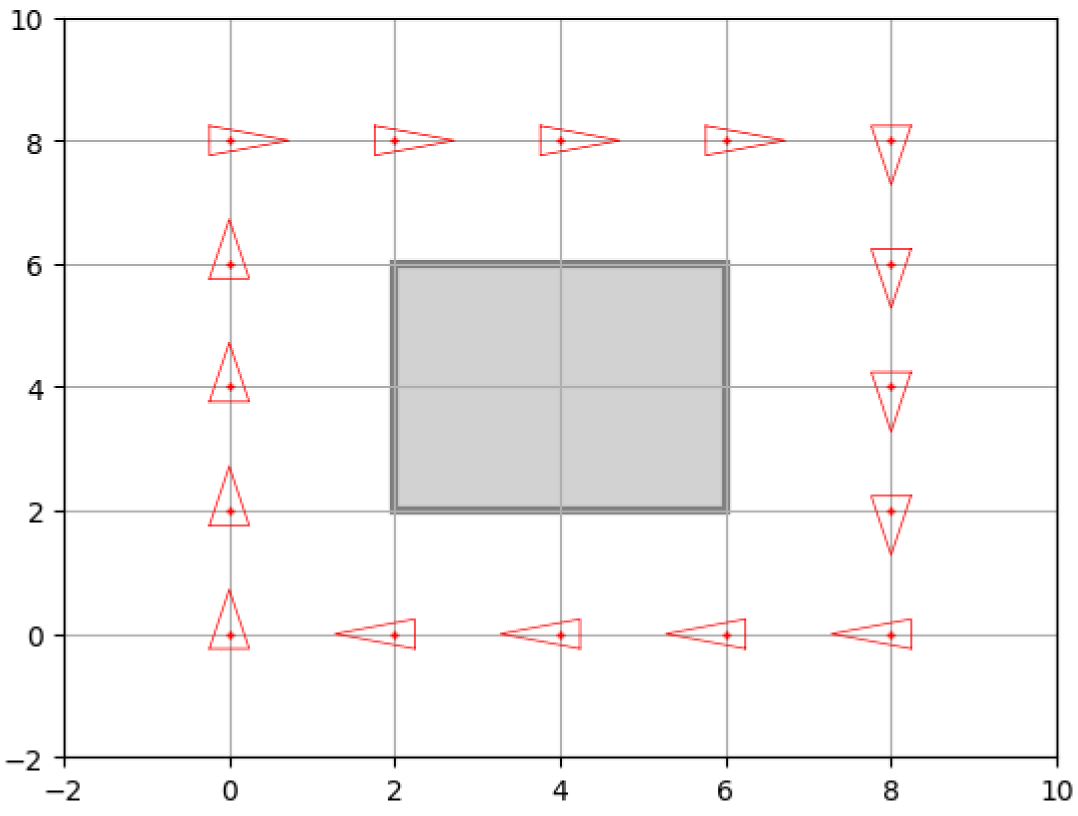
```
In [8]: class Robot():
            '''Mobile robot implementation

               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 [9]: 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='gray',

    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 turn
        # and accordingly set the third component (rotation) of the motion comma
        if not step % turning == 0:
            u[2] = 0
        else:
            u[2] = angle_inc

        # 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 [10]:
```
# 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 and the motion itselft 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 $\Delta p_{\text{given}}$.
- The covariance $\Sigma_{\Delta p}$ is a $3 \times 3$ matrix, which defines the amount of error at each step (time interval). It looks like this:

$$\Sigma_{\Delta p} = \begin{bmatrix} \sigma_x^2 & 0 & 0 \\ 0 & \sigma_y^2 & 0 \\ 0 & 0 & \sigma_\theta^2 \end{bmatrix}$$

To gain insight into the vocariance matrix, let's suppose that we've commanded Giraff to move two meters forward, one to the left, and turns pi/2 to the left a total of twenty times, and we've measured its final position. This is the result:

In [11]:
```
# Array of motion measurements [x_i,y_i,theta_i]
data = np.array([
    [1.9272377,  0.61826959, 1.56767043],
    [2.32512511, 1.00742,    1.5908133],
    [2.18640042, 0.98655067, 1.68010124],
    [1.98890723, 0.96641266, 1.5478623],
    [2.0729443,  0.82685635, 1.67115959],
    [1.975565,   1.20433306, 1.62406736],
     [1.88160001, 1.17310891, 1.54204513],
     [2.21991591, 0.92045473, 1.55294863],
     [1.79006882, 0.97170525, 1.60347324],
     [2.13932179, 1.17665025, 1.57022972],
     [1.89099453, 0.86546558, 1.52364342],
     [1.78903666, 0.93264142, 1.60133537],
     [2.05418773, 1.34436849, 1.58577607],
     [2.12027142, 1.15626879, 1.5552685],
     [2.04842395, 1.22015604, 1.58246969],
     [2.00209448, 0.77744971, 1.55656092],
     [2.06276761, 0.88401541, 1.62989382],
     [1.70384096, 1.12819609, 1.61440142],
     [1.84918712, 1.26022099, 1.50058668],
     [2.02138316, 1.12614774, 1.52156016]
])
```

## *ASSIGNMENT 2: Calculating the covariance matrix*

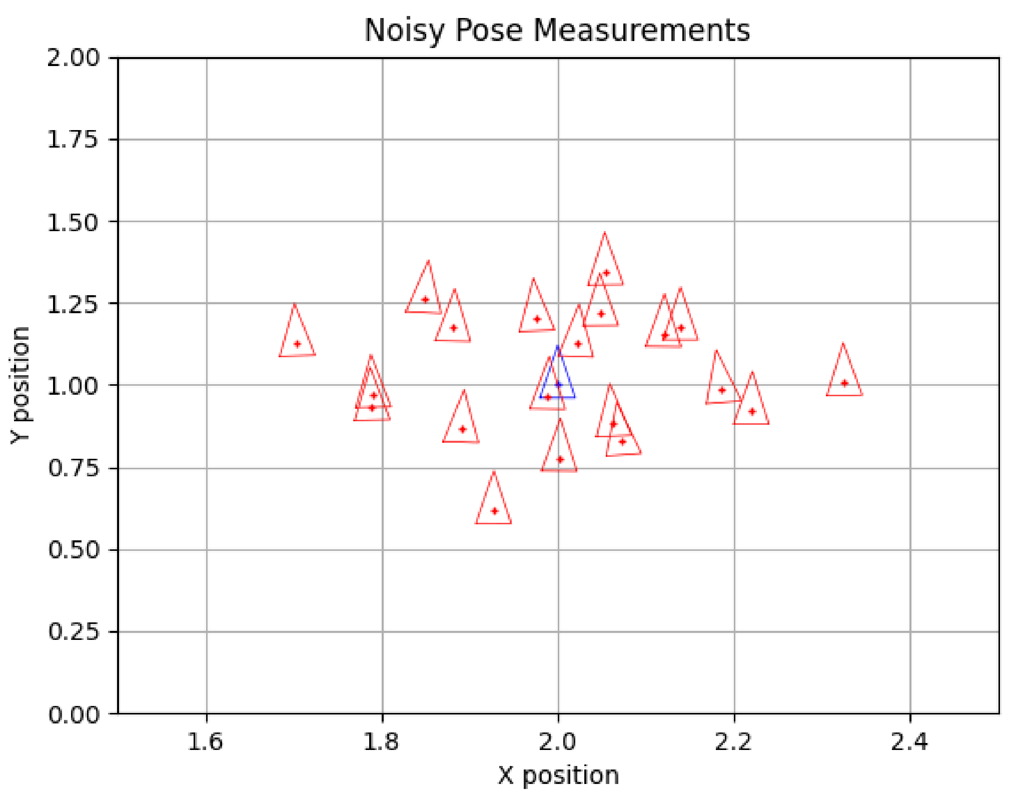
Complete the following code to compute the covariance matrix characterizing the motion uncertainty of the Giraff robot. Ask yourself what the values in the diagonal mean, and what happens if they increase/decrease.

*Hints:* `np.var()` , `np.diag()`

In [12]:
```
# Compute the covariance matrix (since there is no correlation, we only compute
cov_x = np.var(data[:, 0])
cov_y = np.var(data[:, 1])
cov_theta = np.var(data[:, 2])

# Form the diagonal covariance matrix
covariance_matrix = np.diag([cov_x, cov_y, cov_theta])

# PRINT COVARIANCE MATRIX
print("Covariance matrix:")
print(covariance_matrix)

# VISUALIZATION
fig, ax = plt.subplots()
plt.xlim((1.5, 2.5))
plt.ylim((0, 2))
plt.grid('on')
# Commanded pose
DrawRobot(fig, ax, np.vstack([2, 1, np.pi/2]), color='blue')
# Noisy poses
for pose in data:
    DrawRobot(fig, ax, np.vstack([pose[0],pose[1],pose[2]]))
plt.xlabel('X position')
plt.ylabel('Y position')
plt.title('Noisy Pose Measurements')
plt.show()
```

```
Covariance matrix:
[[0.02342594 0.         0.        ]
 [0.         0.03246639 0.        ]
 [0.         0.         0.00212976]]
```

Figure



Expected results:

```
Covariance matrix:
[[0.02342594 0.         0.        ]
 [0.         0.03246639 0.        ]
 [0.         0.         0.00212976]]
```

## *ASSIGNMENT 3: 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()`*, *`stats.multivariate_normal.rvs()`*.
- 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**

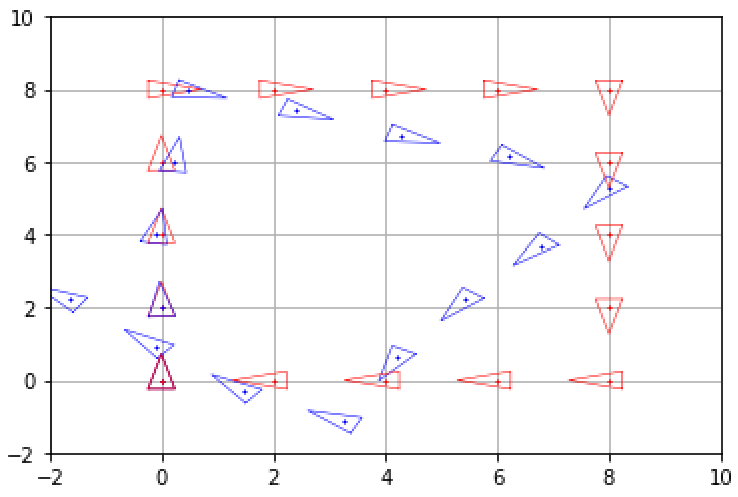
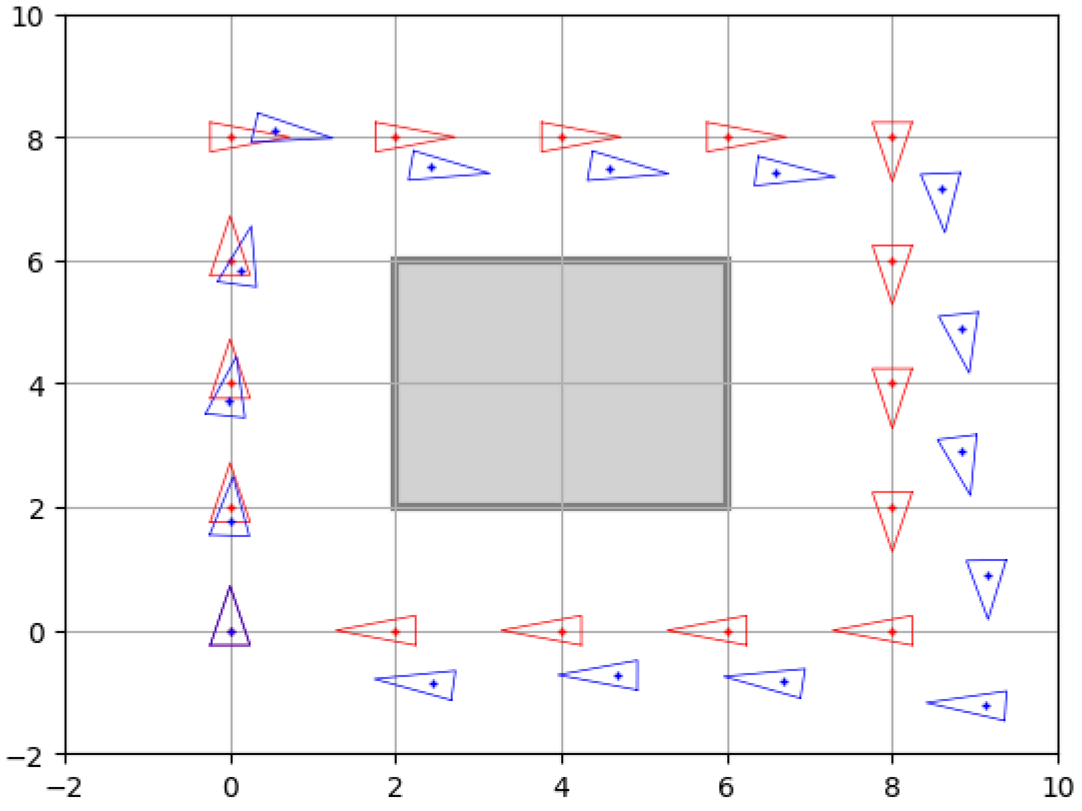


Fig. 3: Movement of our robot using pose compositions. Containing the expected poses (in red) and the true pose affected by noise (in blue)

In [13]:
```python
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 ammount
            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.covarian
            Then this noisy increment is applied to self.true_pose
        """
        super().step(step_increment)
        true_step = np.vstack(stats.multivariate_normal.rvs(step_increment.flatt
        self.true_pose = tcomp(self.true_pose, true_step)

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

In [14]:
```python
# 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?

  *Porque en el mundo real se produce "ruido" al movimiento del robot, como son los errores de calibración o ruedas malfuncionantes. A medida que el robot se mueve, los errores de precisión se van acumulando y el robot no sigue el camino esperado.*

- In which scenario could they be the same?

  *En aquel contexto en el que no haya ningún error de precisión al moverse (no ocurre nada de lo anterior).*

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

  *Los valores de la diagonal principal de la matriz de covarianza nos indican cómo de exacto es este movimiento. El primer valor nos indica la posición en el eje X, el segundo la posición en el eje Y y el tercero su ángulo de rotación. Si estos valores crecen, significa que habrá más inexactitud a la hora de establecer la siguiente pose.*