3.1 Motion through pose composition

November 6, 2020

1 3.1 Motion through pose composition

A fundamental aspect of the development of mobile robots is the motion itself. 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.

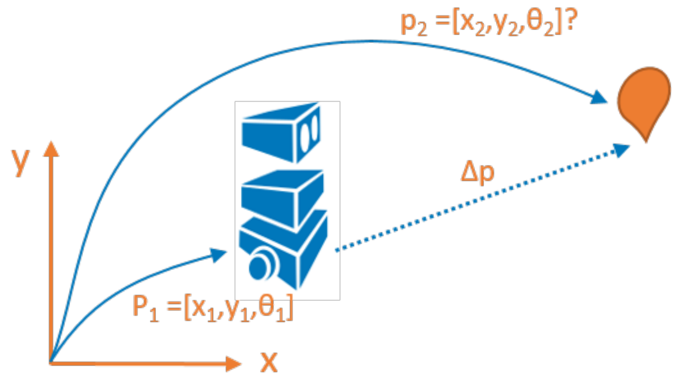


Fig. 1: Example of an initial 2D robot pose (p_1) and its resultant pose (p_2) after completing a motion (Σp).

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.

[1]:
```
%matplotlib notebook

# IMPORTS

import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

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

1.1 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

1.2 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, *i.e.* how much the robot has moved during an interval of time, we compute the final pose p using the **composition of poses** function:

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

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

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.

1.3 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.

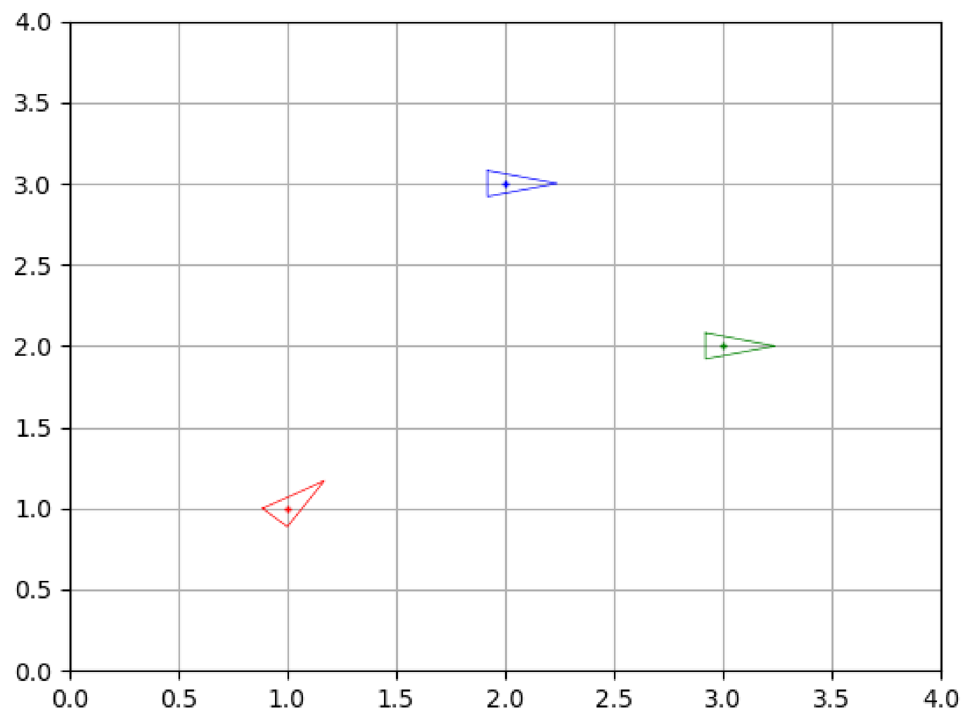
[2]:
```
def composition(p1,p12):

    p2 = np.vstack([0.,0.,0.])

    p2[0,0] = p1[0,0] + p12[0,0]*np.cos(p1[2, 0]) - p12[1,0]*np.sin(p1[2,0])
    p2[1,0] = p1[1,0] + p12[0,0]*np.sin(p1[2, 0]) + p12[1,0]*np.cos(p1[2,0])

    p2[2,0] = p1[2,0] + p12[2,0]

    # Work with angle on range [0, 2*pi] rad
    p2[2,0] = p2[2,0] % (2*np.pi)

    return p2
```

[3]:
```
def inverseComposition(p1, p2):

    p12 = np.vstack([0.,0.,0.])

    p12[0,0] =  (p2[0,0]-p1[0,0])*np.cos(p1[2, 0]) + (p2[1,0]-p1[1,0])*np.
↪sin(p1[2,0])
    p12[1,0] = -(p2[0,0]-p1[0,0])*np.sin(p1[2, 0]) + (p2[1,0]-p1[1,0])*np.
↪cos(p1[2,0])

    p12[2,0] = p2[2,0] - p1[2,0]

    # Work with angle on range [0, 2*pi] rad
    p12[2,0] = p12[2,0] % (2*np.pi)

    return p12
```

[4]:
```
def inversePose(p):

    p_ = np.vstack([0.,0.,0.])

    p_[0,0] =  -p[0,0]*np.cos(p[2,0]) - p[1,0]*np.sin(p[2,0])
    p_[1,0] =   p[0,0]*np.sin(p[2,0]) - p[1,0]*np.cos(p[2,0])

    p_[2,0] = (-p[2,0]) % (2*np.pi) # Work with angle on range [0, 2*pi] rad

    return p_
```

[5]:
```
def compInv(p1,p2):
    return composition(inversePose(p1),p2)
```

[7]:
```
####################################
# First, execute the class Robot #
####################################

p1 = np.vstack([1, 1, np.pi/4])
p12 = np.vstack([3*np.sqrt(2)/2, np.sqrt(2)/2, 2*np.pi-np.pi/4])
p23 = np.vstack([1, -1, 0])

p2 = composition(p1, p12)
p3 = composition(p2, p23)

p13 = compInv(p1, p3) #?

p1_ = inversePose(p1)
p2_ = inversePose(p2)
p3_ = inversePose(p3)

robot = Robot(p1)
robot2 = Robot(p2)
robot3 = Robot(p3)

# VISUALIZATION
fig, ax = plt.subplots()
plt.ion()
plt.draw()
plt.xlim((0, 4))
plt.ylim((0, 4))
plt.grid()

robot.draw(fig, ax, "red")
robot2.draw(fig, ax, "blue")
robot3.draw(fig, ax, "green")

print("p1 -> ", p1.flatten(), "\np2 -> ", p2.flatten(),"\np3 -> ", p3.
 ↪flatten(),"\n")
print("p12 -> ", p12.flatten(), "\np13 -> ", p13.flatten(),"\np23 -> ", p23.
 ↪flatten(),"\n")
print("(-p1 -> inverse of p1)\n")
print("-p1 -> ", p1_.flatten(), "\n-p2 -> ", p2_.flatten(),"\n-p3 -> ", p3_.
 ↪flatten())
print("\nNull operation is satisfied:\ncomposition(-p1,p1) = ", composition(p1_,␣
 ↪p1).flatten())
print("\nNull operation is satisfied:\ncompInv(p1,p1) = ", compInv(p1, p1).
 ↪flatten())
print("\nNull operation is satisfied:\ninverseComposition(p1,p1) = ",␣
 ↪inverseComposition(p1, p1).flatten())
```



```
p1 ->  [1.          1.          0.78539816]
p2 ->  [2. 3. 0.]
p3 ->  [3. 2. 0.]

p12 ->  [2.12132034 0.70710678 5.49778714]
p13 ->  [ 2.12132034 -0.70710678  5.49778714]
p23 ->  [ 1 -1  0]

(-p1 -> inverse of p1)

-p1 ->  [-1.41421356  0.          5.49778714]
-p2 ->  [-2. -3.  0.]
-p3 ->  [-3. -2.  0.]

Null operation is satisfied:
composition(-p1,p1) =  [-1.11022302e-16 -3.33066907e-16  0.00000000e+00]

Null operation is satisfied:
compInv(p1,p1) =  [-1.11022302e-16 -3.33066907e-16  0.00000000e+00]

Null operation is satisfied:
inverseComposition(p1,p1) =  [0. 0. 0.]
```

END OF OPTIONAL PART

1.3.1 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

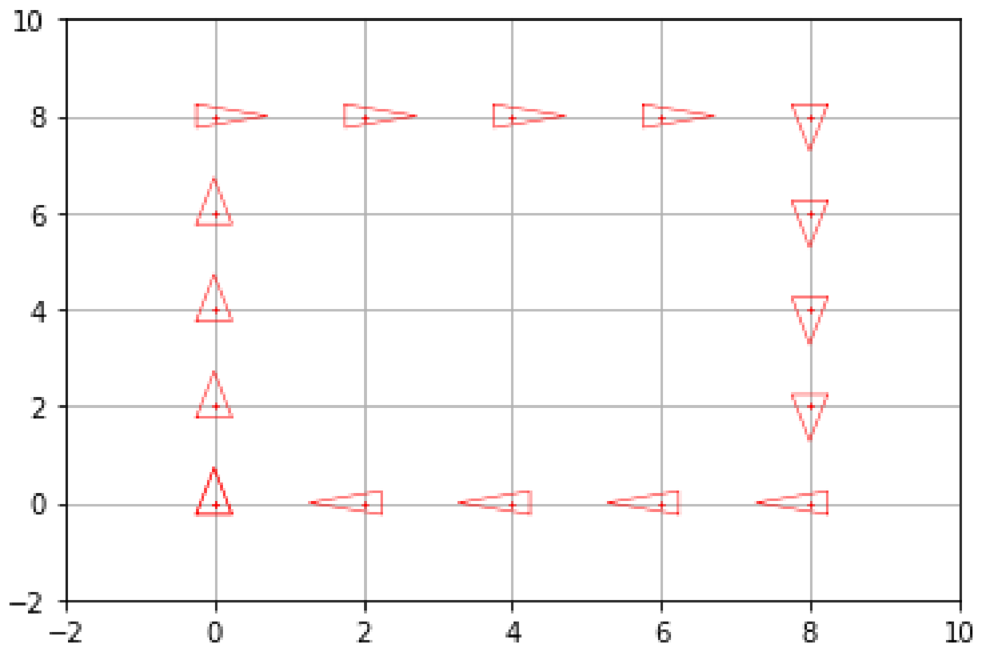


Fig. 2: Route of our robot.

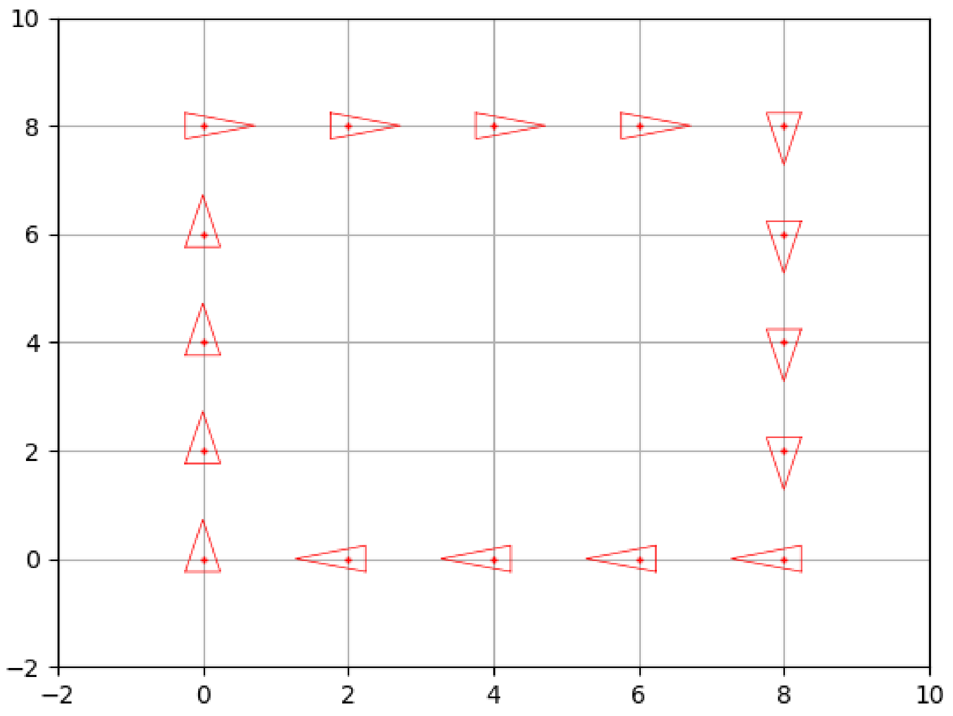
```
[6]: class Robot():
         '''Mobile robot implementation

            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, colour="red"):
             DrawRobot(fig, ax, self.pose, color=colour)
```

[8]:
```
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.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␣
↪command

        turning = turning -1

        if turning == 0:
            u[2] = u[2]+angle_inc
            turning = 4
        else:
            u[2]=0


        # Execute the motion command
        robot.step(u)

        # VISUALIZATION
        robot.draw(fig, ax)
        fig.canvas.draw()
        plt.pause(0.2)
```

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

[9]:
```
# RUN
initial_pose = np.vstack([0., 0., np.pi/2])
```

```
robot = Robot(initial_pose)
main(robot)
```



1.4 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 (multi-dimensional) 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).

1.4.1 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()`, `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

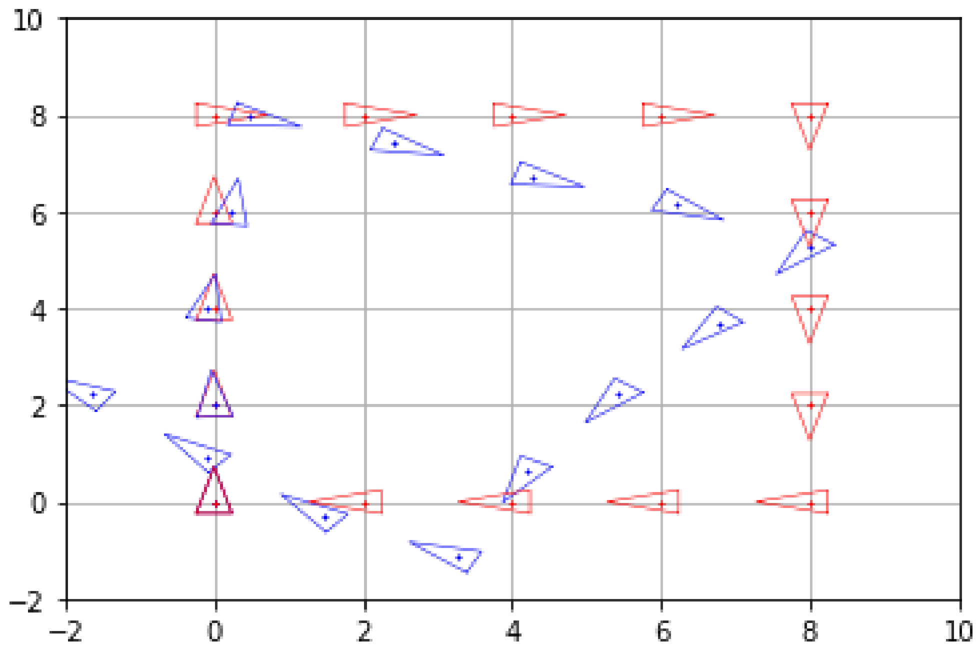
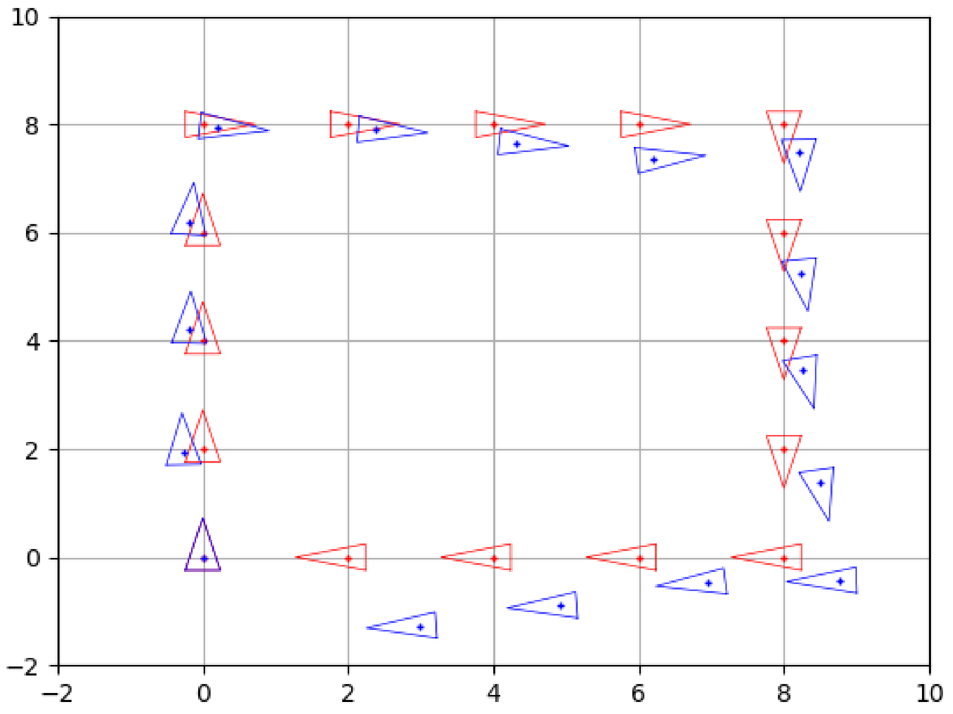


Fig. 3: Movement of our robot using pose compositions. Containing the expected poses (in red) and the true pose affected by noise (in blue)

[10]:
```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␣
 ↪of noise.
            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.
↪covariance.
            Then this noisy increment is applied to self.true_pose
        """
        super().step(step_increment)

        true_step = stats.multivariate_normal.rvs(step_increment.flatten(), self.
↪covariance)
        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')
```

[21]:
```
# 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)
```



1.4.2 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 (blue) and true (red) poses different?

  Because the red pose is the expected and without errors pose and the blue one have some noise added.

- In which scenario could they be the same?

  In the one which the random variable never introduces noise. (Pretty lucky)

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

  When the diagonal values of the covariance matrix are higher, more dispersed could be our robot pose.

3.2 Velocity-based motion model

November 6, 2020

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

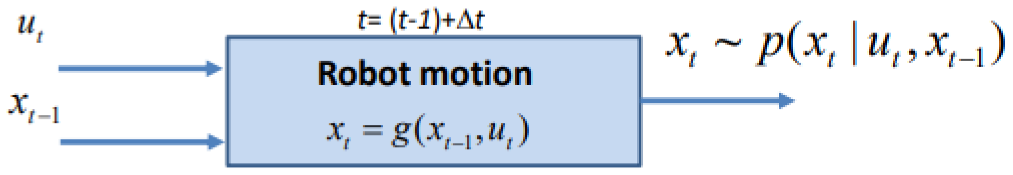


Fig. 1: Inputs and outputs of a probabilistic motion model.

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

1.1 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:

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

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

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

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*
- $\Delta\theta = w \cdot \Delta t$

[2]:
```
%matplotlib notebook

# IMPORTS
import numpy as np
from numpy import random
import matplotlib.pyplot as plt

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

1.1.1 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

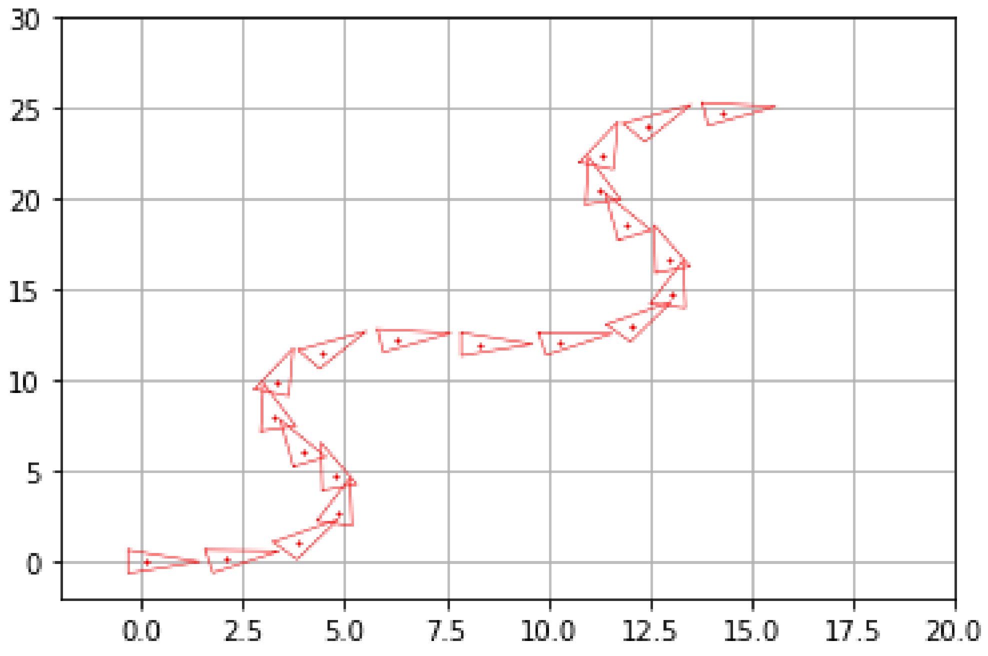


Fig. 2: Route of our robot.

```
[3]: def next_pose(x, u, dt, cov=None):
         ''' This function takes pose x and transform it according to the motion␣
      ↪u=[v,w]'
             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 gaussian noise
         '''
         if cov is not None:
```

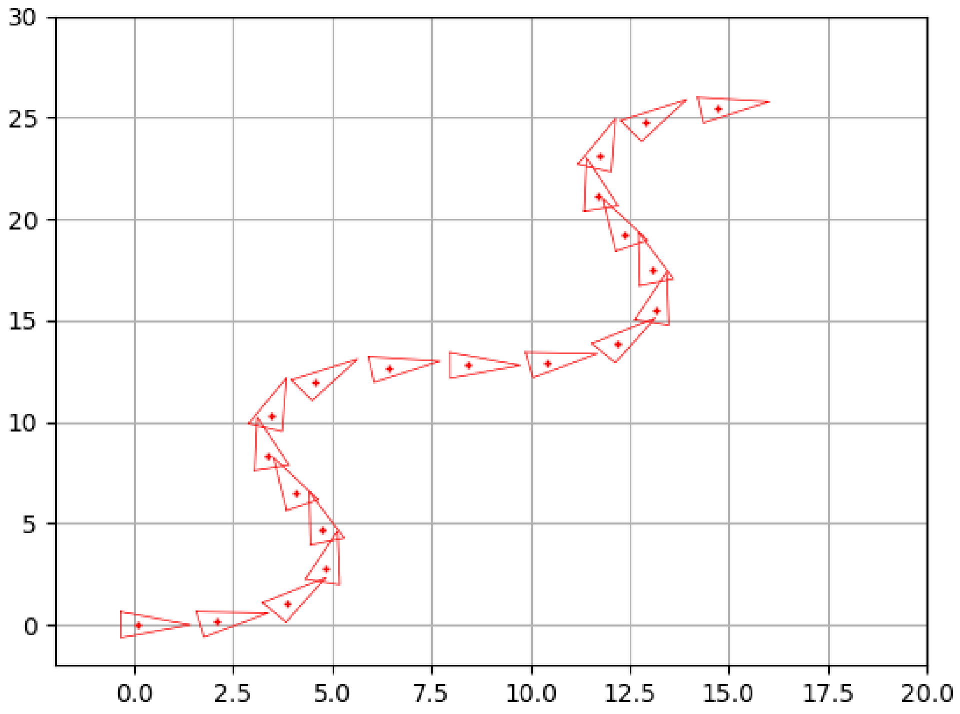
```
        u += np.sqrt(cov) @ random.randn(2, 1)
        #u = np.random.multivariate_normal(u.flatten(),cov)

    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

        dTheta = u[1] * dt
        next_x = np.vstack([x[0] + (-R*np.sin(x[2]) + R*np.sin(x[2] + dTheta)),
                            x[1] + (R*np.cos(x[2]) - R*np.cos(x[2] + dTheta)),
                            x[2] + dTheta])

    return next_x
```

[4]:
```
class VelocityRobot(object):
    """ Mobile robot implementation that uses velocity commands.

        Attr:
            pose: expected pose of the robot in the real world (without taking␣
↪account noise)
            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.

[5]:
```
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.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)
            fig.canvas.draw()
            plt.pause(0.1)
```

[6]:
```
# RUN
dT = 0.1 # time steps size
pose = np.vstack([0., 0., 0.])

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



1.2 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:

$$\Sigma_{u_t} = \begin{bmatrix} \sigma_v^2 & 0 \\ 0 & \sigma_w^2 \end{bmatrix}$$

1.3 OPTIONAL

Write a Markdown cell containing the Jacobians ecuations aforementioned.

END OF OPTIONAL PART

1.3.1 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`.

```
[7]: def next_covariance(x, P, Q, u, dt):
         ''' Compute the covariance of a robot following the velocity motion model

             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 previous␣
      ↪robot heading
         si, ci = np.sin(u[1, 0]*dt), np.cos(u[1, 0]*dt) #sin and cos for the heading␣
      ↪increment
         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*sx],
            [0,1, v*dt*cx],
            [0,0,1]
        ])
        JacF_u = np.array([
            [dt*cx, dt*sx, 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_x = ( JacF_x @ P @ JacF_x.T )

    Pt_u = ( JacF_u @ Q @ JacF_u.T )

    Pt =  Pt_x + Pt_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

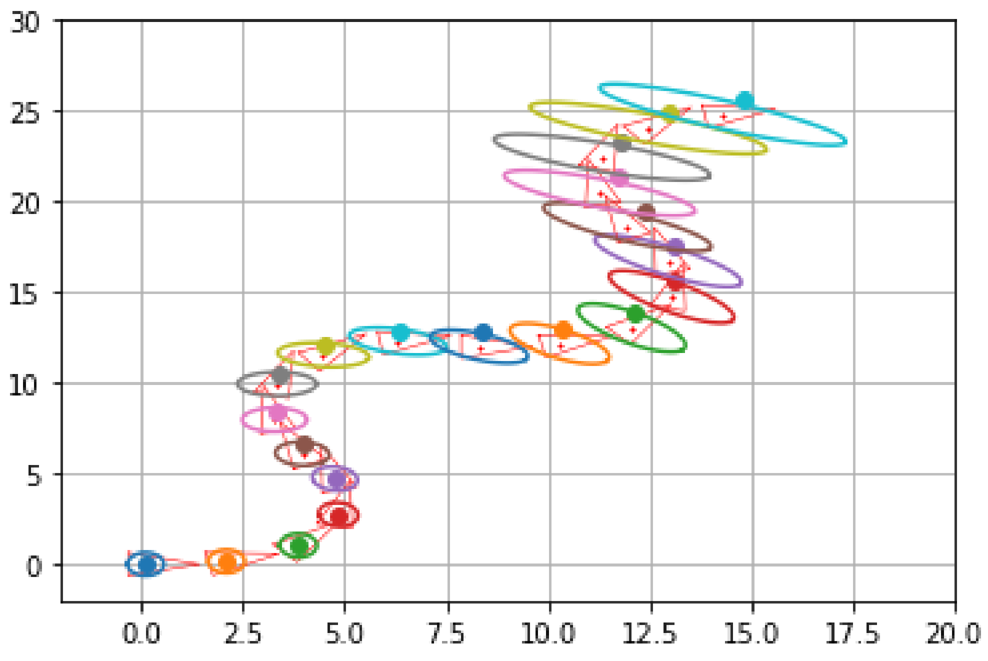


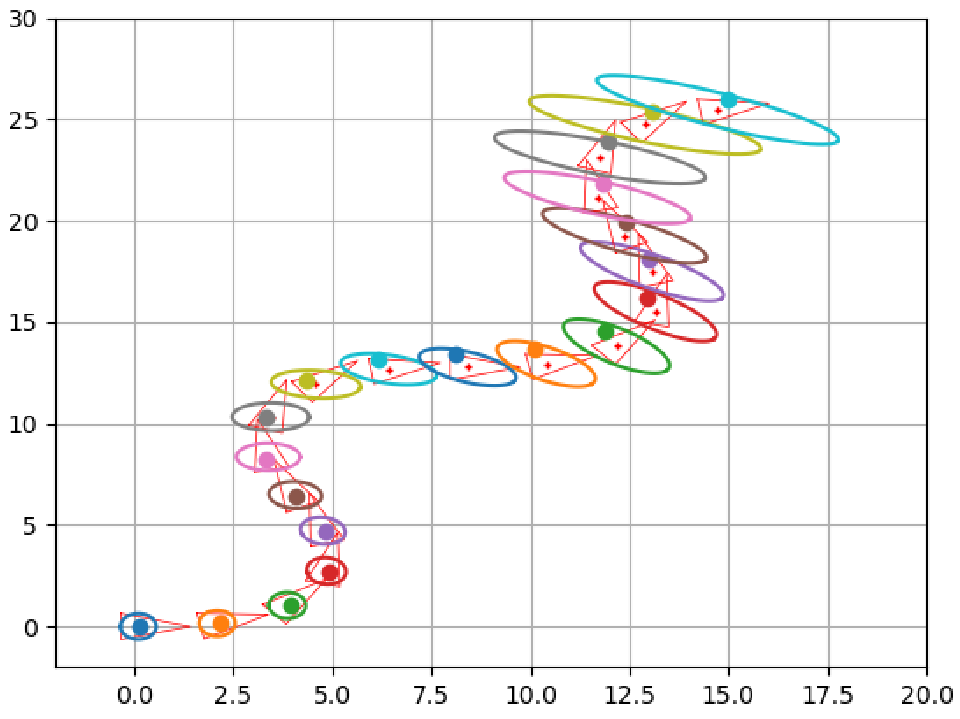
Fig. 3: Movement of a robot using velocity commands. Representing the expected pose (in red), the true pose (as dots) and the confidence ellipse.

```
[8]: 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 (noisy)
                 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, self.cov_move,␣
      ↪u, self.dt)

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

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

Now, **try your implementation!**

[27]:
```
# 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)
```



1.3.2 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?

  Because we need to considerate the possibility that the robot is doing linear moves (w=0) or if it doesn't. For each case, the way to compute the next pose is different.

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

  Two. Linear velocity (v) and angular velocity (w).

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

  We use Jacobians to keep track of how confident we are of the robot's pose. (That is, robot's pose now follows a gaussian distribution).

- 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 we increase SigmaV and/or SigmaW, the true pose of the robot is more distant from the perfect one. This distance increases as steps are taken. If we decrease those values, the true pose is near the perfect pose. If SigmaV and SigmaW are 0, both poses are equal in each step.

3.3 Odometry-based motion model

November 12, 2020

1 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.

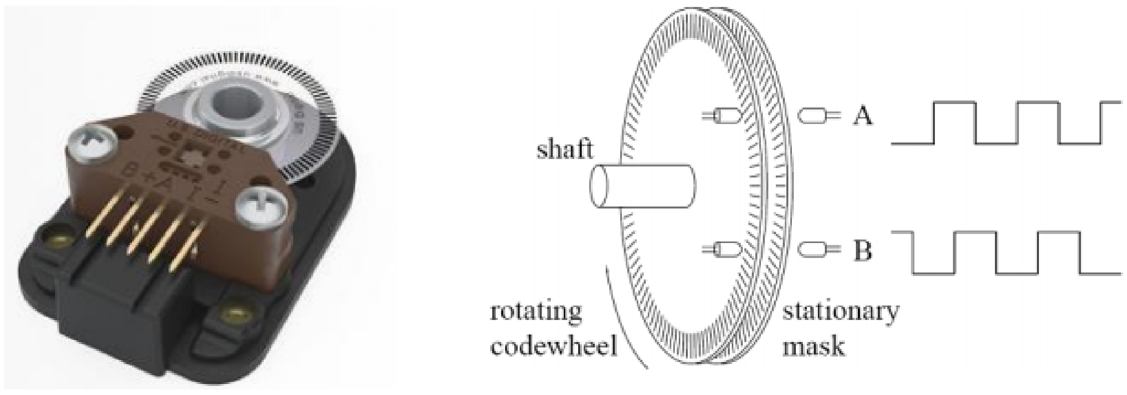


Fig. 1: Example of a wheel encoder used to sum pulses and compute the robot pose.

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, and makes it available to the robot at lower rate (*e.g.* 10Hz) using a tool that we already know: the composition of poses.

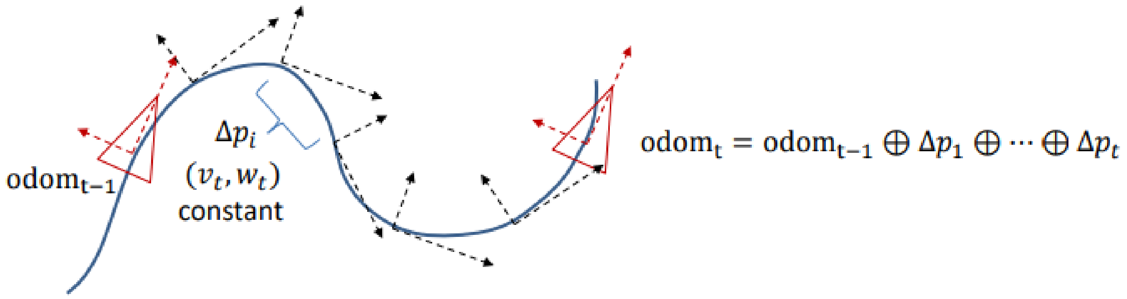

odom_{t-1}
Δp_i
(v_t, w_t)
constant
$\mathrm{odom}_t = \mathrm{odom}_{t-1} \oplus \Delta p_1 \oplus \cdots \oplus \Delta p_t$

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

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: $u_t = [\Delta x_t, \Delta y_t, \Delta\theta_t]^T$ - Sample form: $u_t = [\theta_1, d, \theta_2]^T$

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*.

[1]:
```
%matplotlib notebook

# IMPORTS
import numpy as np
from numpy import random
import matplotlib.pyplot as plt
from scipy import stats

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
```

1.1 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\partial g_{\overline{\partial x_{k-1}=}}

$$\begin{bmatrix} 1 & 0 & -\Delta x_k \sin\theta_{k-1} - \Delta y_k \cos\theta_{k-1} \\ 0 & 1 & \Delta x_k \cos\theta_{k-1} - \Delta y_k \sin\theta_{k-1} \\ 0 & 0 & 1 \end{bmatrix}$$

$, , , , , , , , , , , , \partial g \overline{\overline{\partial u_k =}}$

$$\begin{bmatrix} \cos\theta_{k-1} & -\sin\theta_{k-1} & 0 \\ \sin\theta_{k-1} & \cos\theta_{k-1} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

\[10pt]

andthecovariancematrixofthismovement(

Σ_{u_t}) is defined as seen below. Typically, it is constant during robot motion:

$\Sigma\_\{u\_t\} =$

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

$$

1.1.1 ASSIGNMENT 1: The model in action

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

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

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

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

Example

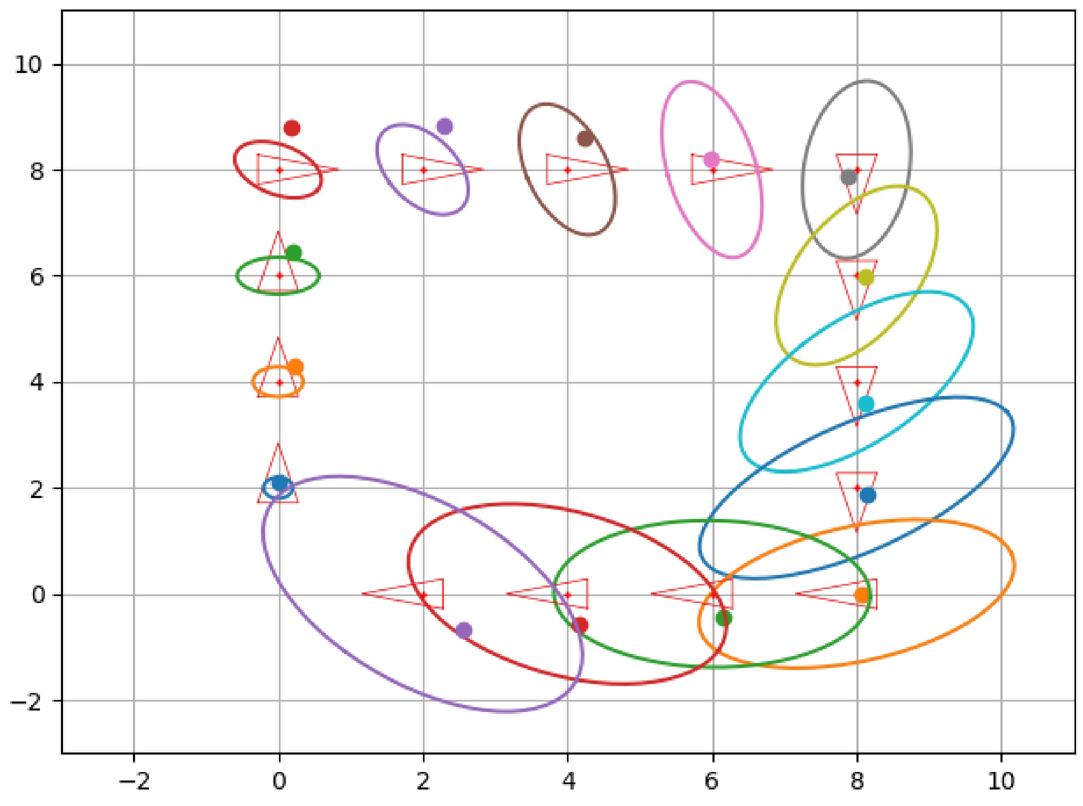


Fig. 2: Movement of a robot using odometry commands. Representing the expected pose (in red), the true pose (as dots) and the confidence ellipse.

```
[18]: 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(prev_pose, u)
```

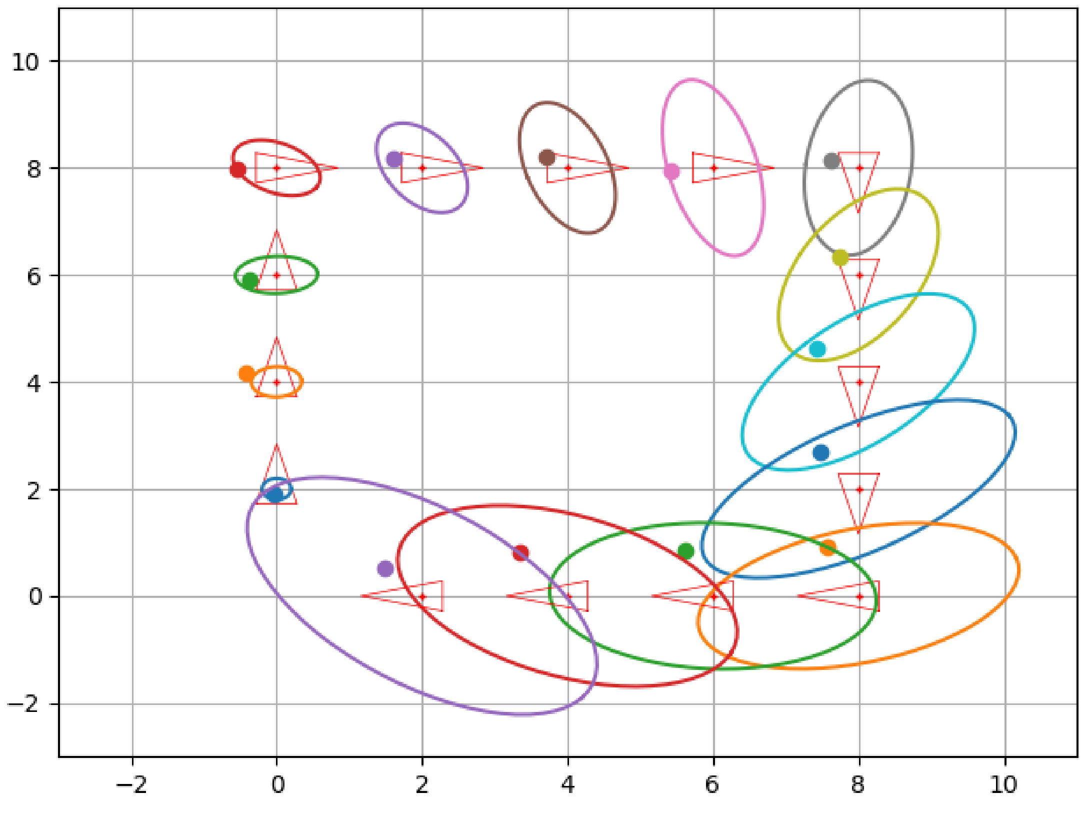
```
        # TODO Generate true pose
        noisy_u = np.vstack(stats.multivariate_normal.rvs(mean=u.flatten(),␣
↪cov=self.Q))
        self.true_pose = tcomp(self.true_pose, noisy_u)

        ## Preguntar ##
        # TODO Update covariance
        JacF_x = J1(prev_pose, u)
        JacF_u = J2(prev_pose, u)

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


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

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

```
[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.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)
        fig.canvas.draw()
        plt.pause(0.3)
```

[19]:
```
## PREGUNTAR ##
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)
```



1.1.2 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 Based motion model, we use the old pose and the velocity of the robot meanwhile in the Odometry Based motion model, we use the old pose and the movement we want to do.

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

  Three. X and Y coords and the angle.

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

  Jacobians are used to obtain the new value of the covariance. (Noise propagated)

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

  If we increase the values of the covariance matrix, the real_pose will probably be more dispersed from the mean. If we decrease them, the real_pose will probably be near the mean.

1.2 3.3.2 Sample form

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

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

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

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

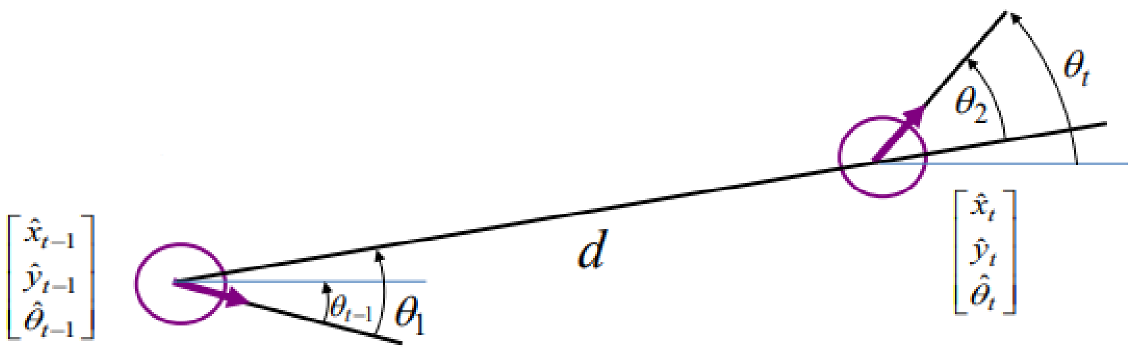

$\begin{bmatrix}\hat{x}_{t-1}\\ \hat{y}_{t-1}\\ \hat{\theta}_{t-1}\end{bmatrix}$
θ_{t-1}
θ_1
d
θ_2
θ_t
$\begin{bmatrix}\hat{x}_t\\ \hat{y}_t\\ \hat{\theta}_t\end{bmatrix}$

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

So this type of order is expressed as:

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

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

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

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

[5]:
```
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]) -␣
 ↪pose_old[2]
    d = np.sqrt((pose_now[1]-pose_old[1])**2 + (pose_now[0]-pose_old[0])**2)
    theta2 = pose_now[2] - pose_old[2] - theta1
    return np.vstack((theta1, d, theta2))
```

Try such function with the code cell below:

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

[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:

$$\theta_1 = \hat{\theta}_1 + \text{sample}\left(\alpha_0 \hat{\theta}_1^2 + \alpha_1 \hat{d}^2\right) d = \hat{d} + \text{sample}\left(\alpha_2 \hat{d}^2 + \alpha_3 \left(\hat{\theta}_1^2 + \hat{d}^2\right)\right) \theta_2 = \hat{\theta}_2 + \text{sample}\left(\alpha_0 \hat{\theta}_2^2 + \alpha_1 \hat{d}^2\right) \quad (2)$$

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:

$$x_t = x_{t-1} + d\cos(\theta_{t-1} + \theta_1)\; y_t = y_{t-1} + d\sin(\theta_{t-1} + \theta_1)\; \theta_t = \theta_{t-1} + \theta_1 + \theta_2 \tag{3}$$

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

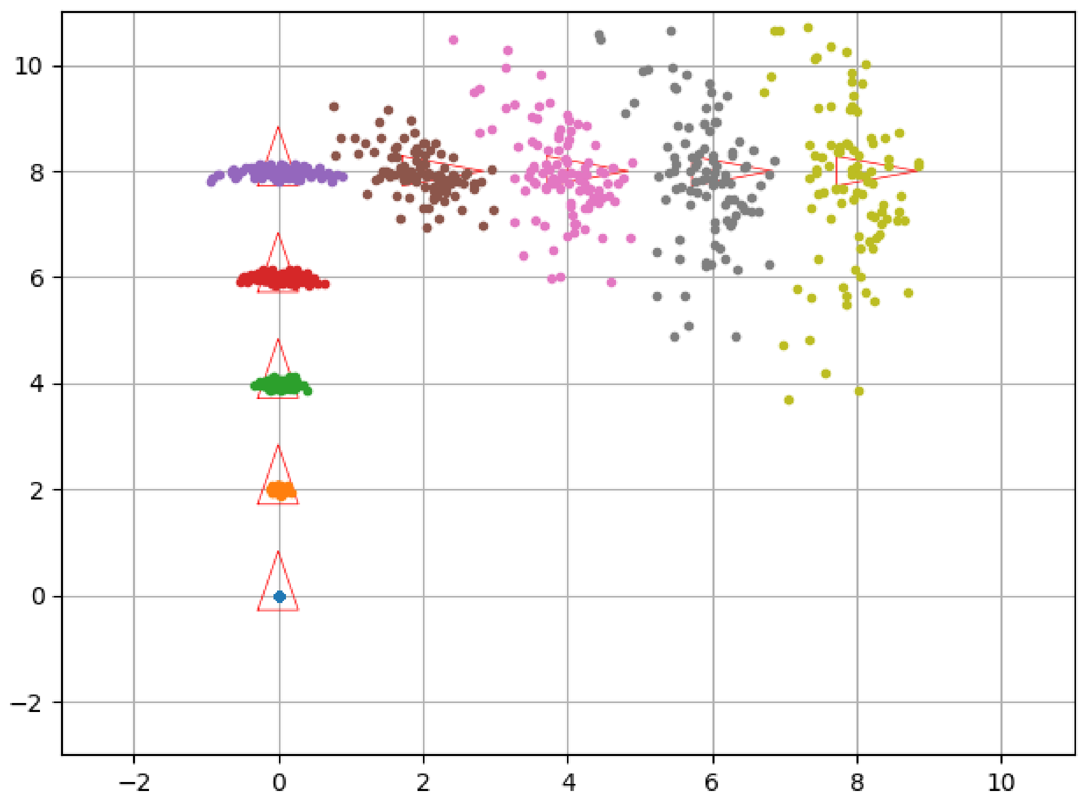


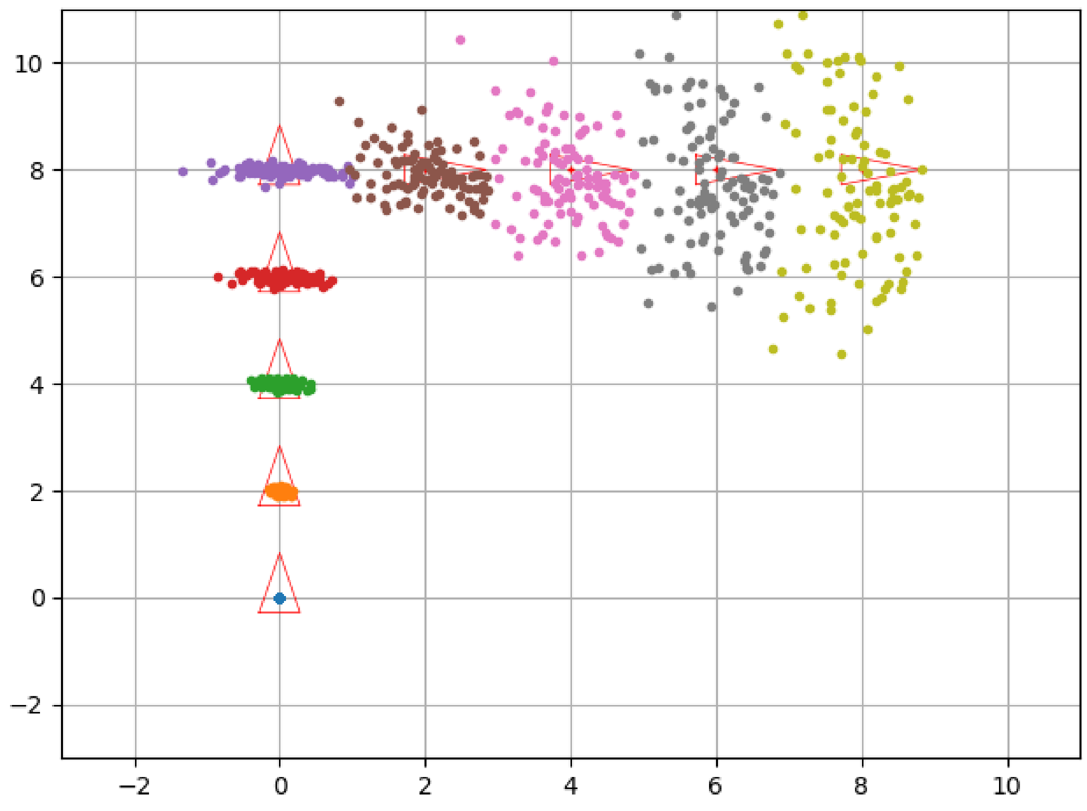
Fig. 1: Movement of a robot using odometry commands in sampling form. Representing the expected pose (in red) and the samples (as clouds of dots)

[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, 0] + u[0, 0]
        self.pose[0, 0] += u[1, 0] * np.cos(ang)
        self.pose[1, 0] += u[1, 0] * np.sin(ang)
        self.pose[2, 0] = u[2, 0] + ang

        # TODO Generate new samples
        sample = lambda b: stats.norm(loc=0, scale=b).rvs(size=self.samples.
↪shape[1])

        u2 = u**2

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

        # 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, :] = noisy_u[2, :] + ang

    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**:

[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)
            fig.canvas.draw()
            plt.pause(0.1)
```

[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)
plt.close();
```



1.2.2 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)?

  As more as we increase a value, more dispersed will the values be

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

  Three. Angle towards the destiny point, distance to destiny point and the desired angle

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

  It have a shape of a half ellipse