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.

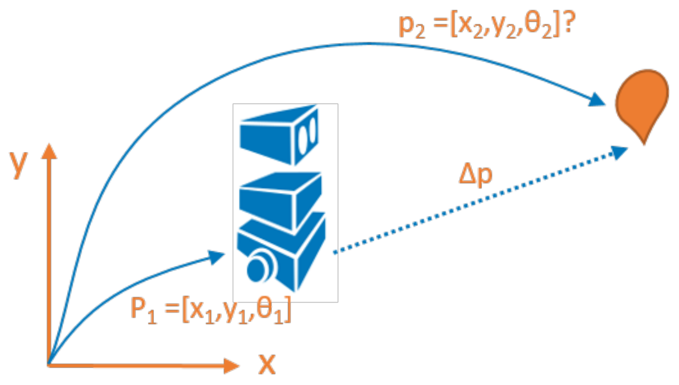


Fig. 1: Example of an initial 2D robot pose (p_1) and its resultant pose (p_2) after completing a motion (Σp).

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.

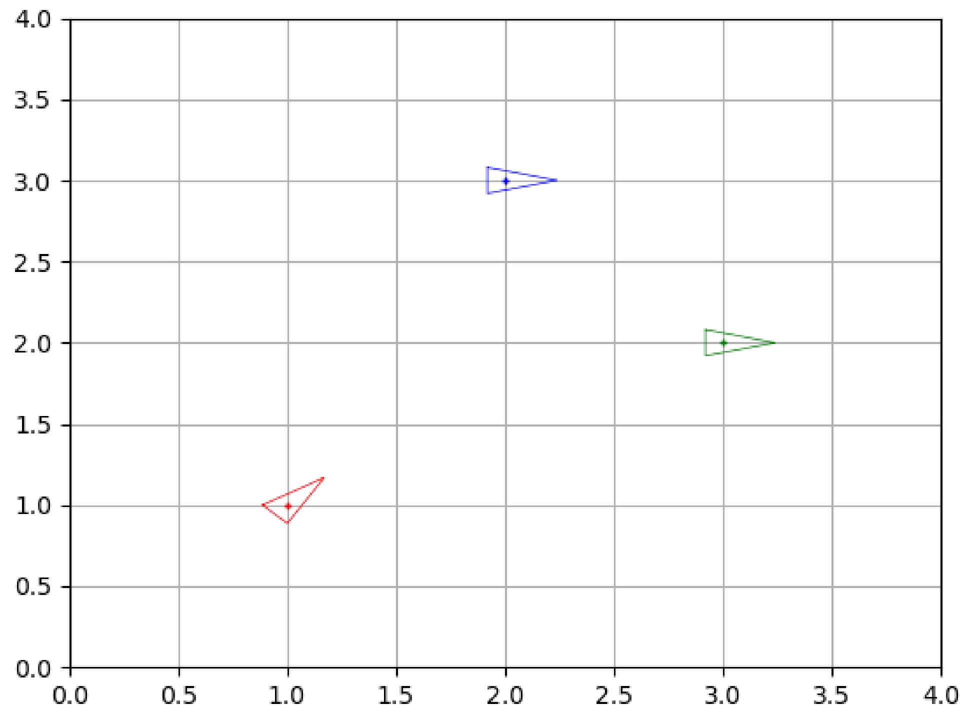
[2]:
```
def composition(p1,p12):

    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

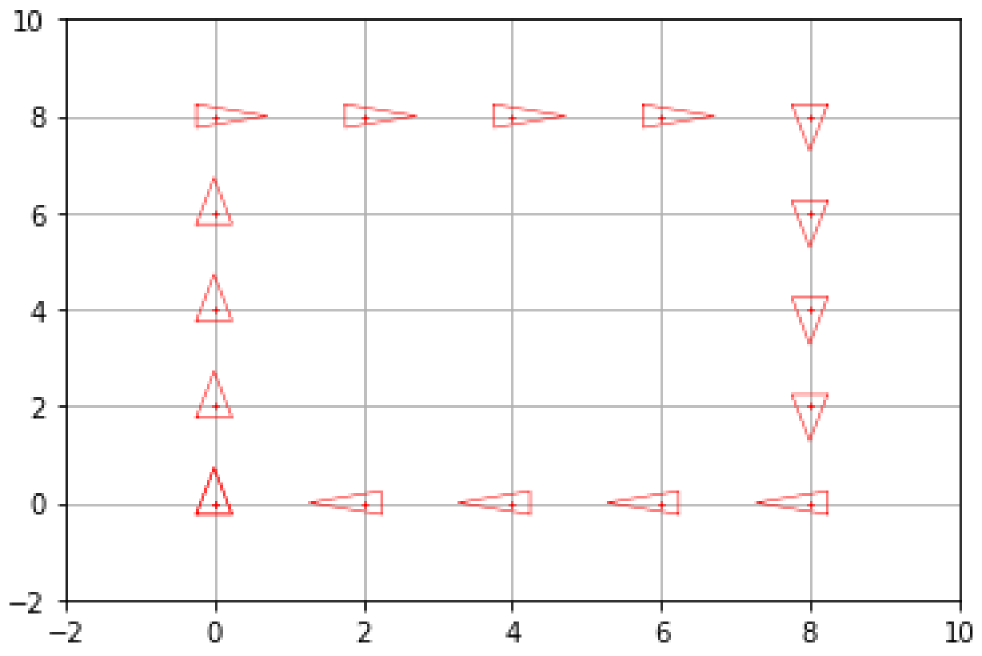


Fig. 2: Route of our robot.

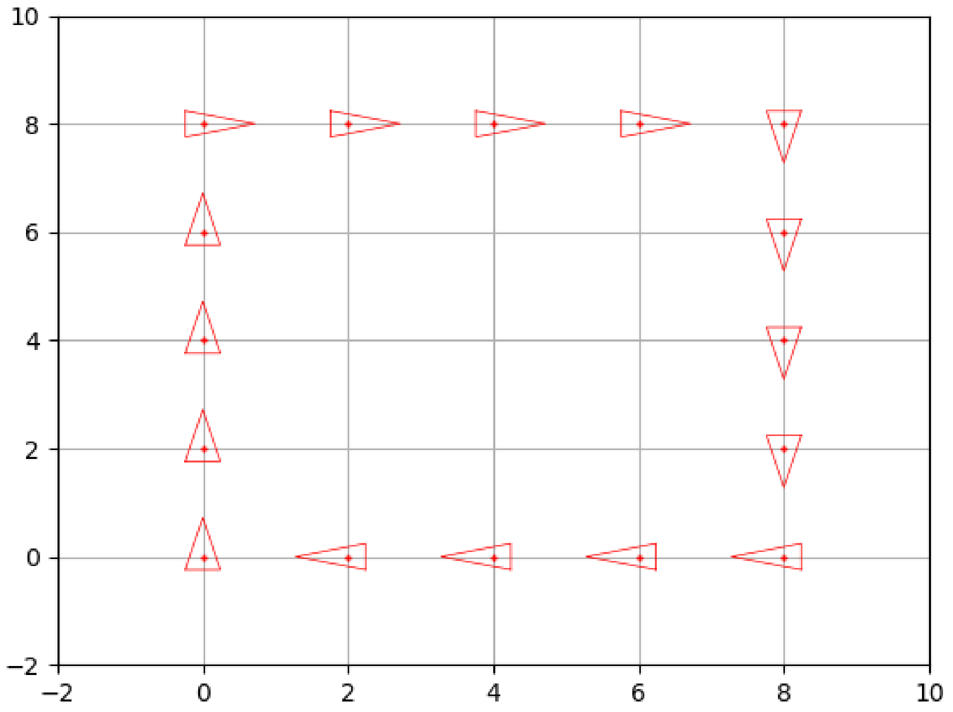
```
[6]: class Robot():
         '''Mobile robot implementation

            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

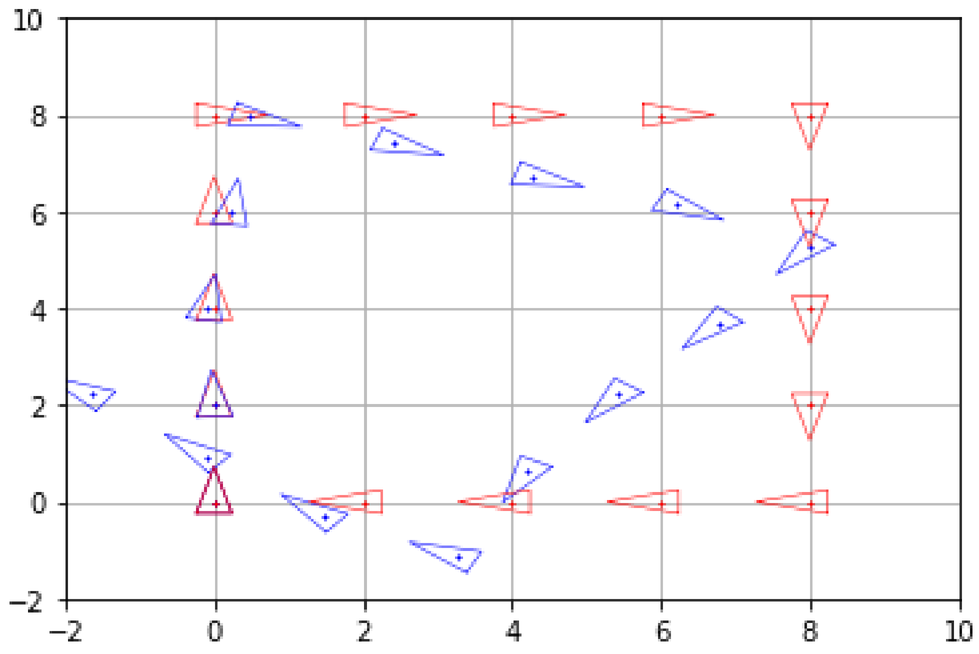
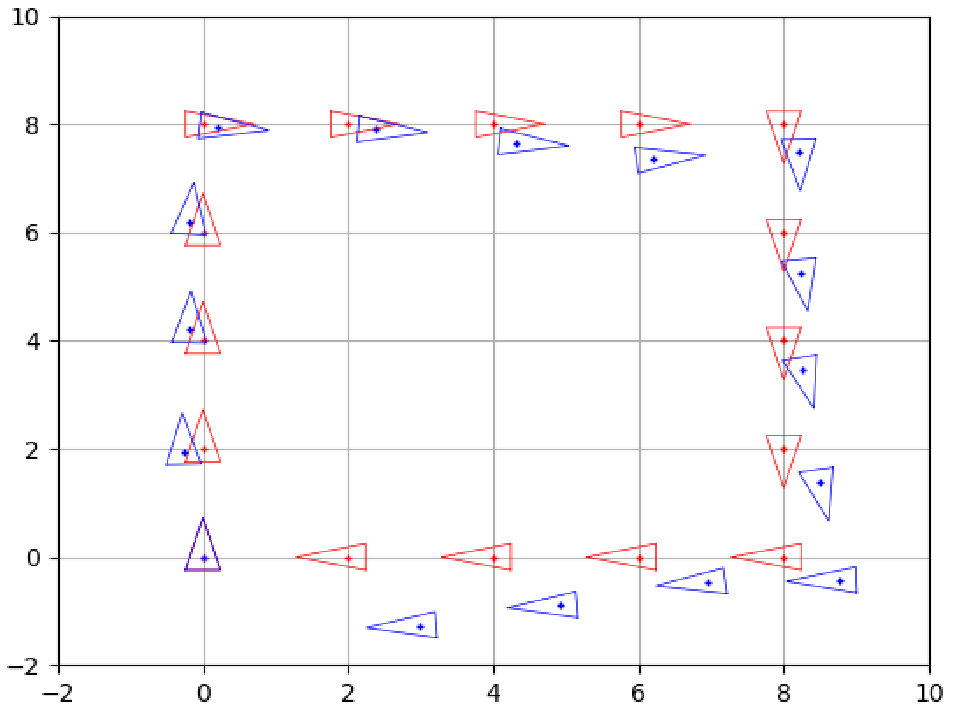


Fig. 3: Movement of our robot using pose compositions. Containing the expected poses (in red) and the true pose affected by noise (in blue)

```
[10]: 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.