A *state* describes the condition of our robot at a specific time. This may include information about where the robot is, or other factors related to the environment that may affect the robot. When we develop a robotics algorithm, we are responsible to decide the information that we wish to include in the state. In other words, the definition of state depends on the task that we are trying to solve. Ideally, we want a state to exhibit the following properties:

1. Markov property: future state is independent of the past given the present:

$$x_{t+1} = f(x_t, x_{t-1}, \ldots, x_0; u_t, \ldots, u_0) = f(x_t; u_t)$$

2. A minimally sufficient statistic for the task (i.e., it contains only the information that we need to solve the task)
3. Permits efficient computation
4. Generalizable

An example of a state representation that is often used in robotics is the *pose*. A *pose* represents the location of the robot in the world as well as its orientation (i.e., what direction is the robot facing towards). Since we live in a 3-dimensional (3D) world, we represent the location of the robot as its (x, y, z) coordinate in the world. We can represent orientation in terms of its roll, pitch, and yaw angles (also known as Euler angle representation). For example, the yaw angle θ_{yaw} will be the angle difference between the longitudinal axis of the robot from the reference frame.

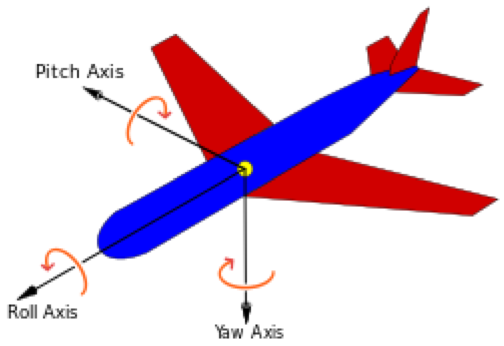


Illustration of roll, pitch, and yaw (source: https://en.wikipedia.org/wiki/Aircraft_principal_axes).

Note that the location and the orientation of the robot can be defined either in the global coordinate frame (i.e., relative to where the origin is) or relative to any other coordinate frames. Combining location and orientation, we can write the pose q in a vector/matrix form:

$$q = [\, x \quad y \quad z \quad \theta_{roll} \quad \theta_{pitch} \quad \theta_{yaw} \,]$$

The pose q can also be represented in the form of a homogeneous transformation matrix, sinces poses are members of the Matrix Lie Group called the **S**pecial **E**uclidean group $SE(3)$:

$$SE(3) = \{T = \begin{bmatrix} R & r \\ 0^T & 1 \end{bmatrix} \in \mathrm{R}^{4\times4} | R \in SO(3), r \in \mathrm{R}^3\},$$

where R denotes the rotation matrix, and r denotes the translation component. This matrix T also has a nice property where

$$T^{-1} = \begin{bmatrix} R^T & -R^T r \\ 0 & 1 \end{bmatrix}$$

One of the reasons to represent the pose in this form is because it allows us to conveniently move between frames (see later in this activity).

Although we have been discussing the pose in $SE(3)$, since we are using a mobile robot and we assume the world is flat, we can simplify the pose of our robot to only its (x, y) coordinate and its yaw (or orientation) angle θ. This is because, in addition to the assumption that the robot is always located at a constant elevation, we can also safely assume roll and pitch angles to always be constant since the robot always touches the ground. So in this case, the pose is simply

$$q = [\, x \quad y \quad \theta \,]$$

, which we can also write in $SE(2)$:

$$SE(2) = \{T = \begin{bmatrix} R & r \\ 0^T & 1 \end{bmatrix} \in \mathrm{R}^{3\times3} | R \in SO(2), r \in \mathrm{R}^2\},$$

where

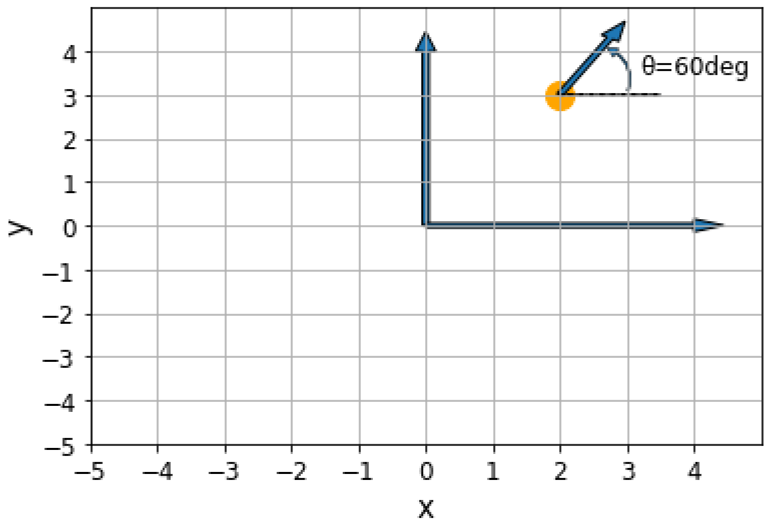
$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \quad r = \begin{bmatrix} x \\ y \end{bmatrix}$$

So, we have:

$$T = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & x \\ \sin(\theta) & \cos(\theta) & y \\ 0 & 0 & 1 \end{bmatrix}$$

Let's first make sure that we understand how to represent a pose by doing the exercise below.

EXAMPLE: representing pose in $SE(n)$



Question:

The figure above illustrates the location of the robot (the orange dot) and its orientation, where the angle is measured from the x-axis (assume the angle to have a positive measure if the rotation is counterclockwise). How do we write the pose in $SE(2)$?

In [ ]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

In [ ]:
```
# Type your answer here
# Tip 1: always express your angles in radians!
# Tip 2: you can go from degrees to radians through the np.deg2rad() functio

theta = np.deg2rad(60)

p = np.array([
    [np.cos(theta), -1*np.sin(theta), 2],
    [np.sin(theta), np.cos(theta), 3],
    [0, 0, 1]
])

print(theta)
print(p)
```

```
1.0471975511965976
[[ 0.5        -0.8660254  2.        ]
 [ 0.8660254  0.5        3.        ]
 [ 0.         0.         1.        ]]
```

(You can find the solutions to this and other questions in the solutions file in this folder. Give it a honest shot yourself before peeking!)

II: Moving between frames

It is often useful to convert a pose from one frame to another one, since we may have multiple coordinate frames used in the description of the same problem. Common examples are robot pose with respect to robot or world frame, or pose of an obstacle with respect to the sensor that perceives it or the robot frame.

As mentioned in the previous section, representing the pose in $SE(n)$ allows us to conveniently move between frames. We can do this by multiplying these transformation matrices together. For example, consider two different poses a and b. If we know the pose a in the origin frame o (i.e., p_a^o) and the pose b in the frame of a (i.e., p_b^a), we can then compute the pose b relative to the origin frame o using the following relation:

$$p_b^o = p_a^o \cdot p_b^a,$$

where p_b^o, p_a^o, and p_b^a are represented in $SE(n)$. Similarly, if we are given p_o^a and p_b^o, we can compute p_b^a via:

$$p_b^a = p_o^a \cdot p_b^o$$

Generally, we can combine these transformations together like the following:

$$p_f^a = p_b^a \cdot p_c^b \cdot p_d^c \cdot p_e^d \cdot p_f^e$$

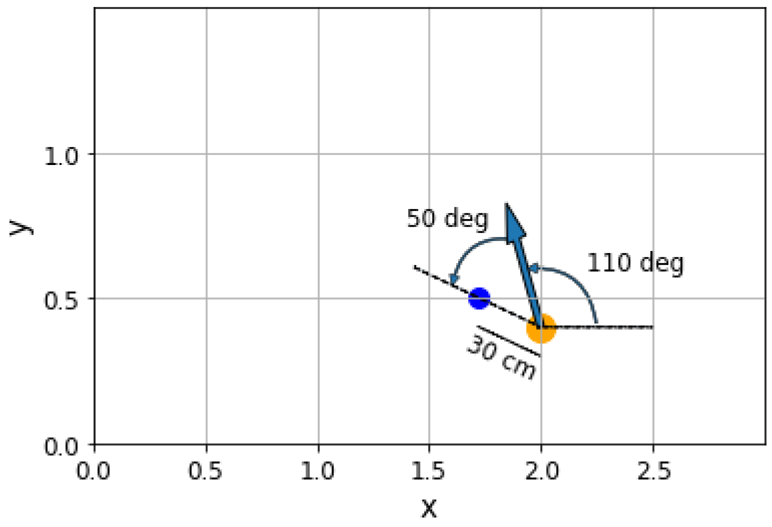
In addition, another handy relation to know is that $p_b^a = (p_a^b)^{-1}$.

Let's now take a look at some examples.

EXAMPLE: moving between frames (from robot frame to global frame)

Your Duckiebot is on a critical recognition mission: identify and place road obstacles on the map.

During its mission, your Duckiebot has GPS support, and it knows its own position in the global (map) frame. It encounters its first obstacle at position $x = 2$ m and $y = 0.4$ m, and orientation θ = 110 degrees. The obstacle itself is at 30cm at 50 degrees (counter-clockwise) from the Duckiebot. Where is the location of the obstacle in the map (i.e., global) frame?



The orange and blue dots represent the robot and the obstacle, respectively. Determine the position of the blue dot in the global frame given the pose of the Duckiebot in the global from and that of the obstacle with respect to the Duckiebot's frame.

In [ ]:
```python
# Run this cell to initialize the problem
# Tip: be consistent with your units – meters and radians are great choices
# Fact: humans understand degrees better than radians. To avoid confusion, i

duckie_pos_g = np.array([2, 0.4])    # Position of Duckiebot in map/global fr
duckie_or_g = 110                    # Orientation of Duckiebot in map/global
obstacle_dist_to_duckie = 0.3        # Obstacle distance to the Duckiebot
obstacle_angle = 50                  # Obstacle angle with respect to Duckieb
```

In [ ]:
```python
# Let's find the answer!

# Step 1. Convert degrees to radians
duckie_or_g_rad = np.deg2rad(duckie_or_g)
obstacle_angle_rad = np.deg2rad(obstacle_angle)

# Step 2. Transformation matrix from Duckiebot (a) to origin (o)
p_o_a = np.array([
    [np.cos(duckie_or_g_rad), -1*np.sin(duckie_or_g_rad), duckie_pos_g[0]],
    [np.sin(duckie_or_g_rad), np.cos(duckie_or_g_rad), duckie_pos_g[1]],
    [0, 0, 1]
])

# Step 3. Transformation matrix from obstacle (b) to Duckiebot (a)
p_a_b =  np.array([
    [np.cos(obstacle_angle_rad), -1*np.sin(obstacle_angle_rad), obstacle_dis
    [np.sin(obstacle_angle_rad), np.cos(obstacle_angle_rad), obstacle_dist_t
    [0, 0, 1]
])

# Step 4. Use the above to compute the transformation from obstacle (b) to o
p_o_b = np.dot(p_o_a, p_a_b)
```

```
# Step 5. Extract the information requested from the transformation matrix
obstacle_pos_g_x = p_o_b[0][2]
obstacle_pos_g_y = p_o_b[1][2]

# Write your answer (position of obstacle in global frame) here instead of "
obstacle_pos_g = np.array([obstacle_pos_g_x, obstacle_pos_g_y])
print(obstacle_pos_g)
```

```
[1.71809221 0.50260604]
```

Correct result: [1.71809221 0.50260604]

EXAMPLE: moving between frames (from global frame to robot frame)

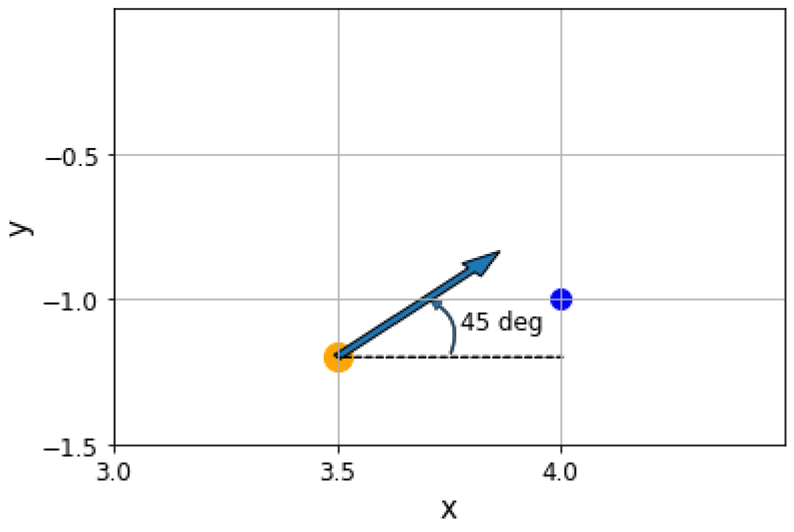
A very concerned duckiecitizen reaches out to you:

"The sky is falling on our heads!"

A sizeable fragment of the citizen neighbour's roof fell on the road. You are told it is positioned at $x = 4$ m and $y = -1$ m (in the global reference frame).

This is valuable information, but before making it official by adding it to the map, it needs to be verified. You have a nearby Duckiebot, at $x = 3.5$ m, $y = -1.2$ m, oriented at $\theta = 45$ degrees, that will be tasked to reach the roof fragment.

What are the coordinates of the obstacle described by the concerned citizen with respect to the Duckiebot's reference frame?



The orange and blue dots represent the robot and the obstacle, respectively. Determine the position of the blue dot in the robot frame.

In [ ]:
```
### Run this cell to initialize the problem
# Tip: remember to be consistent with your units

duckie_pos_g = np.array([3.5, -1.2])   # Position of Duckiebot in global fra
```

```
duckie_or_g = 45                          # Orientation of Duckiebot in global
obstacle_pos_g = np.array([4, -1])        # Position of obstacle in global fram
```

In [ ]:
```
# Let's send the Duckiebot to the correct place!

# Convert degrees to radians
duckie_or_g_rad = np.deg2rad(duckie_or_g)

# Write the transformation matrix from Duckiebot (a) to origin (o)
p_o_a = np.array([
    [np.cos(duckie_or_g_rad), -1*np.sin(duckie_or_g_rad), duckie_pos_g[0]],
    [np.sin(duckie_or_g_rad), np.cos(duckie_or_g_rad), duckie_pos_g[1]],
    [0, 0, 1]
])

# Write the transformation matrix from obstacle (b) to origin (o)
p_o_b = np.array([
    [np.cos(obstacle_pos_g), -1*np.sin(obstacle_pos_g), obstacle_pos_g[0]],
    [np.sin(obstacle_pos_g), np.cos(obstacle_pos_g), obstacle_pos_g[1]],
    [0, 0, 1]
])

# Leverage the known properties of transformation matrices and basics of lir
# inv(p_o_a) = p_a_o => p_a_o * p_o_b = p_a_b
p_a_b = np.dot(np.linalg.inv(p_o_a), p_o_b)

# Extract the obstacle position from the transformation matrix
obstacle_x_a = p_a_b[0][2]
obstacle_y_a = p_a_b[1][2]

# Place your answer here instead of None, None (position of obstacle in robc
obstacle_pos_r = np.array([obstacle_x_a, obstacle_y_a])
print(obstacle_pos_r)
```

```
[ 0.49497475 -0.21213203]
```

Correct result: [0.49497475 -0.21213203]

You can now proceed to the wheel calibration tutorial.

Credit: Rey Reza Wiyatno