```python
"""
Taken from: https://github.com/AtsushiSakai/PythonRobotics/blob/master/Control/move_to_pose/move_to_pose.py

Move to specified pose

Author: Daniel Ingram (daniel-s-ingram)
        Atsushi Sakai (@Atsushi_twi)
        Seied Muhammad Yazdian (@Muhammad-Yazdian)

P. I. Corke, "Robotics, Vision & Control", Springer 2017, ISBN 978-3-319-54413-7

"""

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

from ilqr import iLQRController

class Angle:
    @staticmethod
    def wrap(theta):
        return ((theta + np.pi) % (2*np.pi)) - np.pi

    @staticmethod
    def iswrapped(theta):
        return (-np.pi <= theta) & (theta < np.pi)

    @staticmethod
    def diff(a, b):
        assert Angle.iswrapped(a).all()
        assert Angle.iswrapped(b).all()
        # np.where is like a conditional statement in numpy
        # but it operates on per element level inside the numpy array
        return np.where(a < b,
                        (2*np.pi + a - b),
                        (a - b))
    @staticmethod
    def dist(a, b):
        # The distance between two angles is minimum of a - b and b - a.
        return np.minimum(Angle.diff(a, b), Angle.diff(b, a))


class PIDController:
    """
    Constructs an instantiate of the PIDController for navigating a
    3-DOF wheeled robot on a 2D plane

    Parameters
    ----------
    Kp_rho : The linear velocity gain to translate the robot along a line
             towards the goal
    Kp_alpha : The angular velocity gain to rotate the robot towards the goal
    Kp_beta : The offset angular velocity gain accounting for smooth merging to
              the goal angle (i.e., it helps the robot heading to be parallel
              to the target angle.)
    """

    def __init__(self, Kp_rho, Kp_alpha, Kp_beta):
        self.Kp_rho = Kp_rho
        self.Kp_alpha = Kp_alpha
        self.Kp_beta = Kp_beta

```

```
    def calc_control_command(self, x, x_goal, theta, theta_goal):
        """
        Returns the control command for the linear and angular velocities as
        well as the distance to goal

        Parameters
        ----------
        x : The current position in 2D
        x_goal : The target position in 2D
        theta : The current heading angle of robot with respect to x axis
        theta_goal: The target angle of robot with respect to x axis

        Returns
        -------
        rho : The distance between the robot and the goal position
        v : Command linear velocity
        w : Command angular velocity
        """

        # Description of local variables:
        # - alpha is the angle to the goal relative to the heading of the robot
        # - beta is the angle between the robot's position and the goal
        #   position plus the goal angle
        # - Kp_rho*rho and Kp_alpha*alpha drive the robot along a line towards
        #   the goal
        # - Kp_beta*beta rotates the line so that it is parallel to the goal
        #   angle
        #
        # Note:
        # we restrict alpha and beta (angle differences) to the range
        # [-pi, pi] to prevent unstable behavior e.g. difference going
        # from 0 rad to 2*pi rad with slight turn

        # Proportional control
        #rho = np.hypot(x_diff, y_diff)
        x_diff = x_goal - x
        dhat = np.array([np.cos(theta), np.sin(theta)])
        x_err = (x_diff @ dhat)

        moving_angle = np.arctan2(x_diff[1], x_diff[0])
        moving_angle_err = Angle.diff(np.asarray(moving_angle),
                                      np.asarray(theta))

        dest_angle_err = Angle.diff(np.asarray(theta_goal),
                                    np.asarray(theta))
        v = self.Kp_rho * x_err
        w =  (self.Kp_alpha * moving_angle_err
              if (np.linalg.norm(x_diff) > 0.001) else
              controller.Kp_beta * dest_angle_err)
        return np.array([v, w])
    def control(self, state, state_goal):
        return self.calc_control_command(state[:2], state_goal[:2], state[2],
                                         state_goal[2])

def rotmat2D(theta):
    return np.vstack([np.hstack([np.cos(theta), -np.sin(theta)]),
                      np.hstack([np.sin(theta), np.cos(theta)])])

def move_to_pose(controller,
                 x_start, y_start, theta_start, x_goal, y_goal, theta_goal,
                 dt = 0.01,
                 # Robot specifications
                 MAX_LINEAR_SPEED = 15,
                 MAX_ANGULAR_SPEED = 7,
                 show_animation = True
```

```python
                ):

    pos_goal = np.array([x_goal, y_goal])
    x = np.array([x_start, y_start])
    theta = theta_start

    x_diff = pos_goal - x

    pos_traj = []

    rho = np.linalg.norm(x_diff)
    while rho > 0.001 and np.abs(Angle.diff(np.asarray(theta_goal),
                                            np.asarray(theta))) > 0.001:
        pos_traj.append(x)

        u = controller.calc_control_command(
            x, pos_goal, theta, theta_goal)
        v = u[0]
        w = u[1]

        if abs(v) > MAX_LINEAR_SPEED:
            v = np.sign(v) * MAX_LINEAR_SPEED

        if abs(w) > MAX_ANGULAR_SPEED:
            w = np.sign(w) * MAX_ANGULAR_SPEED

        theta = Angle.wrap(theta + w * dt)
        x = x + v * np.array([np.cos(theta), np.sin(theta)]) * dt
        x_diff = pos_goal - x
        rho = np.linalg.norm(x_diff)

        if show_animation:  # pragma: no cover
            plt.cla()
            plt.arrow(x_start, y_start, np.cos(theta_start),
                      np.sin(theta_start), color='r', width=0.1)
            plt.arrow(x_goal, y_goal, np.cos(theta_goal),
                      np.sin(theta_goal), color='g', width=0.1)
            plot_vehicle(x[0], x[1], theta,
                         [p[0] for p in pos_traj],
                         [p[1] for p in pos_traj],
                        dt=dt)


def plot_vehicle(x, y, theta, x_traj, y_traj, dt):  # pragma: no cover
    # Corners of triangular vehicle when pointing to the right (0 radians)
    p1_i = np.array([0.5, 0, 1]).T
    p2_i = np.array([-0.5, 0.25, 1]).T
    p3_i = np.array([-0.5, -0.25, 1]).T

    T = transformation_matrix(x, y, theta)
    p1 = np.matmul(T, p1_i)
    p2 = np.matmul(T, p2_i)
    p3 = np.matmul(T, p3_i)

    plt.plot([p1[0], p2[0]], [p1[1], p2[1]], 'k-')
    plt.plot([p2[0], p3[0]], [p2[1], p3[1]], 'k-')
    plt.plot([p3[0], p1[0]], [p3[1], p1[1]], 'k-')

    plt.plot(x_traj, y_traj, 'b--')

    # for stopping simulation with the esc key.
    plt.gcf().canvas.mpl_connect(
        'key_release_event',
        lambda event: [exit(0) if event.key == 'escape' else None])

```

```python
    plt.xlim(0, 20)
    plt.ylim(0, 20)

    plt.pause(dt)


def transformation_matrix(x, y, theta):
    return np.array([
        [np.cos(theta), -np.sin(theta), x],
        [np.sin(theta), np.cos(theta), y],
        [0, 0, 1]
    ])

def unicycle_f(x_t, u_t, dt):
    theta = x_t[2]
    return np.array([x_t[0] + u_t[0] * np.cos(theta) * dt,
                     x_t[1] + u_t[0] * np.sin(theta) * dt,
                     x_t[2] + u_t[1] * dt])

def unicycle_Jf_x(x_t, u_t, dt):
    theta = x_t[2]
    return np.array([
        [1., 0., - u_t[0] * np.sin(theta) * dt],
        [0., 1.,   u_t[0] * np.cos(theta) * dt],
        [0., 0.,  1.]
    ])

def unicycle_Jf_u(x_t, u_t, dt):
    theta = x_t[2]
    return np.array([
        [ np.cos(theta) * dt, 0.],
        [ np.sin(theta) * dt, 0.],
        [  0., dt]
    ])



def main():
    # simulation parameters
    dt = 0.01
    pid_controller = PIDController(9, 15, 3)
    lqr_controller = iLQRController(
        Q = np.diag([0.9, 0.9, 0.1]),
        R = np.eye(2) * 0.01,
        f = unicycle_f,
        Jf_x = unicycle_Jf_x,
        Jf_u = unicycle_Jf_u,
        dt = dt,
        init_controller = pid_controller)
    controller = lqr_controller

    for i in range(5):
        x_start = 20 * random()
        y_start = 20 * random()
        theta_start = 2 * np.pi * random() - np.pi
        x_goal = 20 * random()
        y_goal = 20 * random()
        theta_goal = 2 * np.pi * random() - np.pi
        print("Initial x: %.2f m\nInitial y: %.2f m\nInitial theta: %.2f rad\n" %
              (x_start, y_start, theta_start))
        print("Goal x: %.2f m\nGoal y: %.2f m\nGoal theta: %.2f rad\n" %
              (x_goal, y_goal, theta_goal))
        move_to_pose(controller,
                     x_start, y_start, theta_start, x_goal, y_goal,
                     theta_goal,
```

```
                     dt = dt)


if __name__ == '__main__':
    main()
```