```python
import numpy as np

def dare_backpropagation(Qs, Rs, As, Bs):
    # Dicrete algebraic Riccati equation
    P_T = Qs[-1]
    Ps = [P_T]
    K_T = np.linalg.solve(Rs[-1] + Bs[-1].T @ P_T @ Bs[-1],
                          Bs[-1].T @ P_T @ As[-1])
    Ks = [K_T]
    for Q, R, A, B in reversed(list(zip(Qs[:-1], Rs, As, Bs))):
        P_Tminus1 = A.T @ P_T @ A - A.T @ P_T @ B @ K_T + Q
        P_T = P_Tminus1
        Ps.append(P_T)
        K_T = np.linalg.solve(R + B.T @ P_T @ B,  B.T @ P_T @ A)
        Ks.append(K_T)

    return list(reversed(Ps)), list(reversed(Ks))

class RandomController:
    def __init__(self, m, minu, maxu):
        self.m = m
        self.minu = minu
        self.maxu = maxu
    def control(self, state, state_goal):
        return np.random.rand(self.m) * (self.maxu - self.minu) + self.minu

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

    def __init__(self, Q, R, f, Jf_x, Jf_u, dt, N=10, T=10,
                 init_controller=None):
        self.Q = Q
        self.R = R
        self.f = f
        self.Jf_x = Jf_x
        self.Jf_u = Jf_u
        self.dt = dt
        self.N = N
        self.T = T
        self.init_controller = RandomController(self.R.shape[0],
                                                -1, 1)

    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
        """
        state_goal = np.hstack((x_goal, theta_goal))
```

```python
        state = np.hstack((x, theta))
        return self.control(state, state_goal)

    def control(self, state, state_goal):
        # make goal the origin
        T = self.T
        controls = []
        states = [state]
        for t in range(T):
            x_t = states[-1]
            u_t = self.init_controller.control(x_t, state_goal)
            states.append(self.f(x_t, u_t, self.dt))
            controls.append(u_t)
        for i in range(self.N): # Refine the trajectory for N iterations
            As = []
            Bs = []
            Qs = []
            Rs = []
            n = 3
            m = 2
            for t in range(T): # unroll trajectory for T time steps
                # Linearize the dynamics around states, controls
                x_lin_t = states[t]
                u_lin_t = controls[t]
                # x_ref_t could be time dependent but here we have a fixed
                # state goal
                x_ref_t = state_goal

                # Get the jacobian around linearization point
                A_t = self.Jf_x(x_lin_t, u_lin_t, self.dt)
                # Construct A matrix for homogeneous state space
                A_t_hom = np.zeros((n+1, n+1))
                A_t_hom[:-1, :-1] = A_t
                A_t_hom[-1, -1] = 1
                A_t_hom[:-1, -1] = self.f(x_lin_t, u_lin_t, self.dt) - x_lin_t
                A_t_hom[-1, :-1] = 0
                As.append(A_t_hom)

                # Construct B matrix for homogeneous state space
                B_t = self.Jf_u(x_lin_t, u_lin_t, self.dt)
                B_t_hom = np.zeros((n+1, m))
                B_t_hom[:-1, :] = B_t
                B_t_hom[-1, :-1] = 0
                Bs.append(B_t_hom)

                # Construct Q matrix for homogeneous state space
                Q_t = self.Q
                Q_t_hom = np.eye(n+1)
                Q_t_hom[:-1, :-1] = Q_t
                Q_t_hom[-1, -1] = (x_lin_t - x_ref_t) @ (x_lin_t - x_ref_t)
                Q_t_hom[:-1, -1] = Q_t @ (x_lin_t - x_ref_t)
                Q_t_hom[-1, :-1] = Q_t_hom[:-1, -1]
                Qs.append(Q_t_hom)

                # Construct R matrix for homogeneous state space
                R_t = self.R
                Rs.append(R_t)

            x_T = states[T]
            Q_T = self.Q
            Q_T_hom = np.eye(n+1)
            Q_T_hom[:-1, :-1] = Q_T
            Q_T_hom[-1, -1] = (x_T - state_goal) @ (x_t - state_goal)
            Q_T_hom[:-1, -1] = Q_T @ (x_T - state_goal)
            Q_T_hom[-1, :-1] = Q_T_hom[:-1, -1]
```

```
            Qs.append(Q_T_hom)

            # Solve for discrete algebraic riccati equation
            Ps, Ks = dare_backpropagation(Qs, Rs, As, Bs)

            # Perturb the trajectory with new  trajectory
            new_states = [state]
            new_controls = []
            for t in range(T):
                x_lin_t = states[t]
                u_lin_t = controls[t]
                x_t = new_states[-1]
                u_t = u_lin_t - Ks[t] @ np.hstack([(x_t - x_lin_t), 1])
                x_tp1 = self.f(x_t, u_t, self.dt)
                new_states.append(x_tp1)
                new_controls.append(u_t)
            controls = new_controls
            states = new_states
        return controls[0]
```