Daniel Vargas Diaz

Homework #4 CS 5844: Human-Robot Interaction

Predict and Blend

1.1

Consider a shared autonomy algorithm that blends the robot action $a_{\mathcal{R}}$ and human input $a_{\mathcal{H}}$ according to:

$$a = (1-\alpha) \cdot a_{\mathcal{H}} + \alpha \cdot a_{\mathcal{R}} \tag{1}$$

where $\alpha \in (0,1)$. The human takes actions that move directly towards the goal and the robot takes actions in the opposite direction with magnitude Δ:

$$a_{\mathcal{H}} = \theta^* - s, \qquad a_{\mathcal{R}} \propto -(\theta^* - s), \qquad \|a_{\mathcal{R}}\| = \Delta \tag{2}$$

Find the distance between the state s and the human's goal θ^* when the system converges. Your answer should be a function of α and Δ.

We know from the question that

$$\Delta = \|a_R\|$$

We could say that the distance between the goal theta and the state s multiply by a constant K is equal to the magnitude of the robot action

$$\Delta = K * \|\theta^* - s\|$$

$$K = \frac{\Delta}{\|\theta^* - s\|}$$

We also know that the system converges when the action taken by the system is equal to 0

$$0 = (1-\alpha) * a_H + \alpha * a_R$$

$$0 = (1-\alpha) * \theta^* - s - \alpha * K * \theta^* - s$$

$$0 = (1-\alpha) * (\theta^* - s) - \alpha * K * (\theta^* - s)$$

$$0 = (\theta^* - s) * (1 - \alpha - \alpha * K)$$

From this we have two equations. If we take the first term, equal to 0, we said that theta equal to s, which does not make sense, this means, that the other term is the one that should be equal to cero

$$0 = (1 - \alpha - \alpha * K)$$

$$0 = (1 - \alpha - \frac{\alpha * \Delta}{\|\theta^* - s\|})$$

$$\|\theta^* - s\| = \frac{\alpha * \Delta}{(1-\alpha)})$$

1.2

Download the provided code predict-blend.py. Here a simulated human may change their goal halfway through the task, and we need to help them reach their goal despite this change. Develop your own approach for predicting the human's goal and assisting the human's actions. Describe your approach using pseudocode. Across 1000 runs, show that your approach results in an average error of < 0.4 units

```python
"""
Starter code for HW4 Problem 1.2

"""

import numpy as np


def predict_goal(s0, st, THETA, prior, beta=1):
    P = [0.] * len(THETA)
    for idx in range(len(THETA)):
        P[idx] = prior[idx]
    dist_so_far = np.linalg.norm(st - s0)
    for idx, theta in enumerate(THETA):
        dist_to_goal = np.linalg.norm(theta - st)
        min_dist = np.linalg.norm(theta - s0)
        num = np.exp(beta * min_dist)
        den = np.exp(beta * (dist_so_far + dist_to_goal))
        P[idx] *= num / den
    P = np.asarray(P)
    return P / np.sum(P)

def get_assist(s, THETA, P):
    aR = np.array([0., 0.])
    for idx, theta in enumerate(THETA):
        atheta = theta - s
        if np.linalg.norm(atheta) > 1.0:
            atheta /= np.linalg.norm(atheta)
        aR += P[idx] * atheta
    return aR

# simulated human
# this human may change their goal midway through the task
def simulated_human(s, t, human_goal, THETA):
    # at timestep t = 10, the human might switch goals
    if t == 10:
        goal_idx = np.random.choice(2)
        human_goal = THETA[goal_idx]
    # human takes noisy action towards chosen goal
    ah = human_goal - s + np.random.normal(0, 0.3, 2)
    if np.linalg.norm(ah) > 1.0:
        ah /= np.linalg.norm(ah)
    return ah, human_goal

# main loop; each run has new goals and a new start position
def main():

    # each run we have two randomly placed goals
    theta1 = 10.0 * np.random.rand(2)
    theta2 = 10.0 * np.random.rand(2)
    THETA = [theta1, theta2]
    human_goal = THETA[np.random.choice(2)]

    # robot starts in a random state
    s = 10.0 * np.random.rand(2)

    # robot starts with uniform belief over the goals
    b = np.array([0.5, 0.5])

    # initialize state
    s0 = np.copy(s)

    # for T=20 timesteps
    #print("Initial Goal: ", human_goal)
    #print("Initial State:", s0)
    for t in range(20):

        # get human action
```

```
        # human_goal is the human's actual goal
        # we only use this to check our approach's accuracy;
        # the robot does not have access to "human_goal"
        ah, human_goal = simulated_human(s, t, human_goal, THETA)
        #print("Human action: ",ah)

        # implement your algorithm here for helping the human
        # your approach should predict the human's goal
        # then take actions to assist
        # remember: you know THETA, but not human_goal
        #p_theta = predict_goal(s0, s, THETA, b)
        #p_theta_all.append(p_theta)
        if(t==10):
          #print("can change now")
          s0 =  np.copy(s)
        if(t<11):
          ar = get_assist(s, THETA, b)
        elif(t>=11):
          p_theta = predict_goal(s0, s, THETA, b)
          ar = get_assist(s, THETA, p_theta)
          if(p_theta[0]>0.6 or p_theta[1]>0.6):
            b = p_theta


        #print("State: ", s)

        # take blended action
        alpha = 0.8 # do not change this value
        a = (1 - alpha) * ah + alpha * ar

        # transition to the next state
        s1 = s + a
        s = np.copy(s1)
    #print("Final Goal: ", human_goal)
    #print("Final state: ", s)
    #print("Error: ", s - human_goal)
    return np.linalg.norm(s - human_goal)

# run the main loop and find average error
total_error = 0.
for _ in range(1000):
    error = main()
    total_error += error
print(total_error / 1000.)
```

```
0.3144225444575468
```

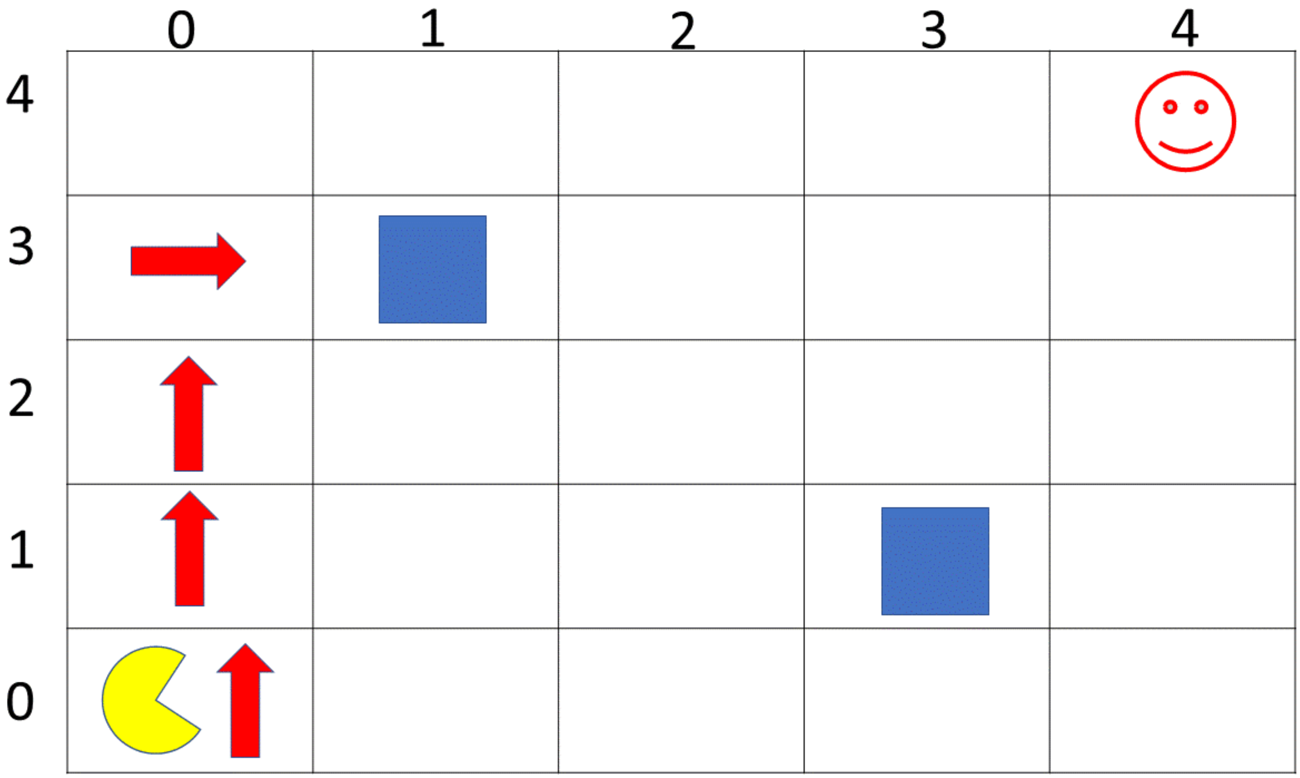
Partially Observable Markov Decision Process

Consider a 5×5 gridworld where the robot starts at $(0,0)$ and the human is standing at $(4,4)$. The robot can move up, down, left, right, *or stay in place*, and the dynamics are deterministic. The robot is trying to quickly rescue an item from the room, but it does not know where the item is. With 70% probability the item is at position $(3,1)$, and with 30% probability the item is at $(1,3)$. The robot can ask the human and remove any uncertainty by moving to position $(4,4)$. The reward at every state besides the item's position is 0.

2.1

Let the reward for reaching the item be +10.0. Is it optimal for the robot to ask to the human if γ = 0.5? Draw the robot's trajectory starting from (0, 0).

In this case, the robot consider that is not optimal to ask the human. This is because the discount factor is small enogh to make the robot only search for the direct reward, which in this case is the most likely reward, the one with 70% of chances.




```
"""
Goal POMDP (example)

"""

import numpy as np


class POMDP:

     # initialization
    def __init__(self):

        # state space
        # here states include x-y position
        # and the belief that the goal is at theta_1
        # b = 1 means we are confident the goal is at theta_1
        # b = 0 means we are confident the goal is at theta_2
        self.states = []
        for s_x in range(0, 5):
            for s_y in range(0, 5):
                for b in np.linspace(0, 1, 4):
                    b = round(b, 1)
                    self.states.append((s_x, s_y, b))

        # action space
        self.actions = ((0,1),(0,-1),(1,0),(-1,0),(0,0))
        # discount factor
        self.gamma = 0.5

    # deterministic dynamics
    def f(self, s, a):

        # take commanded action
        next_states = {}
```

```
        s1 = self.take_action(s, a)

        # get human input
        # this occurs at a specific state
        # only at this state do we get insight from the
        # human about which goal to go for :)
        if s[0] == 4 and s[1] == 4:
            s1_goal1 = (s1[0], s1[1], 1.0)
            s1_goal2 = (s1[0], s1[1], 0.0)
            # 70% of the time is going to be theta_1
            # 30% of the time is going to be theta_2
            next_states[s1_goal1] = 0.7
            next_states[s1_goal2] = 0.3
        else:
            next_states[s1] = 1.0

        return next_states

    # helper function for dynamics
    def take_action(self, s, a):
        # s1 = s + a, belief stays constant
        s1 = (s[0] + a[0], s[1] + a[1], s[2])
        # keeps robot inside state space
        if s1 in self.states:
            return s1
        else:
            return s

    # reward for goal 1
    # this goal is at position (3, 1)
    def reward1(self, s):
        if s[0] == 3 and s[1] == 1:
            return +10.0
        else:
            return 0.0

    # reward for goal 2
    # this goal is at position (1, 3)
    def reward2(self, s):
        if s[0] == 1 and s[1] == 3:
            return +10.0
        else:
            return 0.0


# Q-function
def Qfunction(mdp, s, a, V):
    next_states = mdp.f(s, a)
    expected_V1 = sum(next_states[s1] * V[s1] for s1 in next_states)
    # get expected reward using belief
    expected_reward = s[2] * mdp.reward1(s) + (1.0 - s[2]) * mdp.reward2(s)
    return expected_reward + mdp.gamma * expected_V1

# get optimal policy from value function
def policy(mdp, V):
    pi = {}
    for s in mdp.states:
        # take action that maximizes Q-function
        max_Q = -np.inf
        for action in mdp.actions:
            Q = Qfunction(mdp, s, action, V)
            if Q > max_Q:
                max_Q = Q
                pi[s] = action
    return pi

# value iteration algorithm to get value function
def value_iteration(mdp, epsilon=0.001):
```

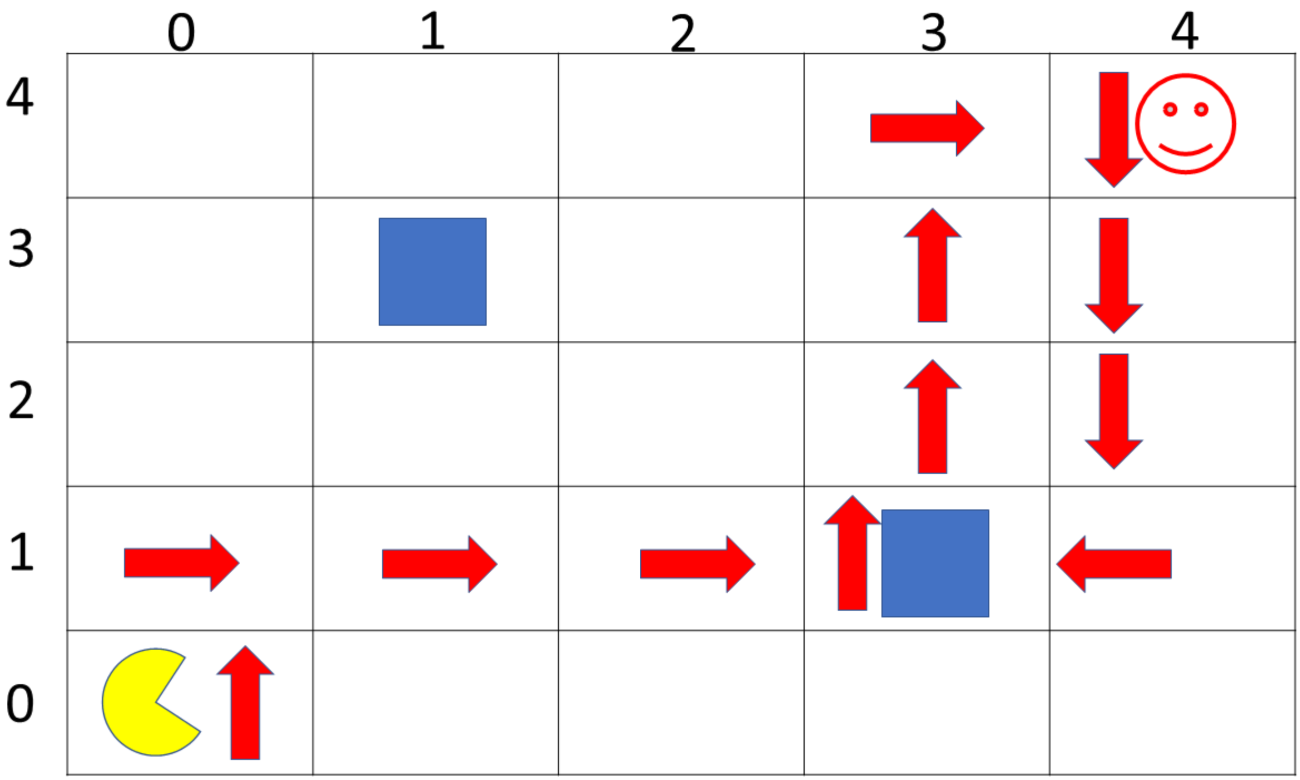
```
    V1 = {s: 0 for s in mdp.states}
    while True:
        V = V1.copy()
        delta = 0
        for s in mdp.states:
            # use value function V to get estimate V'
            action_values = []
            for action in mdp.actions:
                next_states = mdp.f(s, action)
                action_value = sum(next_states[s1] * V[s1] for s1 in next_states)
                action_values.append(action_value)
            # get expected reward using belief
            expected_reward = s[2] * mdp.reward1(s) + (1.0 - s[2]) * mdp.reward2(s)
            V1[s] = expected_reward + mdp.gamma * max(action_values)
            # find largest change in value
            delta = max(delta, abs(V1[s] - V[s]))
        # stop if largest change is small enough
        if delta <= epsilon * (1 - mdp.gamma) / mdp.gamma:
            return V

def main():

    # setup the pomdp
    # this functions the same as an mdp
    # but now with belief as part of state
    robot = POMDP()

    # use value iteration to get V
    V = value_iteration(robot)

    # get optimal policy from V
    pi = policy(robot, V)

    # print policy
    print(pi)
    # rollout policy from chosen start state
    s = (0, 0, 0.3)
    for t in range(20):
        a = pi[s]
        print(s, a)
        next_states = robot.f(s,a)
        # sorry this part is confusing
        # need to pick the next state
        s = list(next_states.keys())[0]

main()
```

```
{(0, 0, 0.0): (0, 1), (0, 0, 0.3): (0, 1), (0, 0, 0.7): (0, 1), (0, 0, 1.0): (0, 1), (0, 1, 0.0): (0, 1), (0, 1, 0.3):
(0, 0, 0.3) (0, 1)
(0, 1, 0.3) (0, 1)
(0, 2, 0.3) (0, 1)
(0, 3, 0.3) (1, 0)
(1, 3, 0.3) (0, 0)
(1, 3, 0.3) (0, 0)
(1, 3, 0.3) (0, 0)
(1, 3, 0.3) (0, 0)
(1, 3, 0.3) (0, 0)
(1, 3, 0.3) (0, 0)
(1, 3, 0.3) (0, 0)
(1, 3, 0.3) (0, 0)
(1, 3, 0.3) (0, 0)
(1, 3, 0.3) (0, 0)
(1, 3, 0.3) (0, 0)
(1, 3, 0.3) (0, 0)
(1, 3, 0.3) (0, 0)
(1, 3, 0.3) (0, 0)
(1, 3, 0.3) (0, 0)
(1, 3, 0.3) (0, 0)
```

2.2

Let the reward for reaching the item be +1.0. Is it optimal for the robot to ask to the human if γ = 0.99? Draw the robot's trajectory starting from (0, 0).

In this case it is optiomal for the robot to ask the human, because the discount factor is big enought to make the robot focus on the long term reward. This results in the robot asking the human and the going to one of the goals.




```
"""
Goal POMDP (example)

"""

import numpy as np


class POMDP:

     # initialization
    def __init__(self):

        # state space
        # here states include x-y position
        # and the belief that the human wants theta_1
        # b = 1 means we are confident human wants theta_1
        # b = 0 means we are confident human wants theta_2
        self.states = []
        for s_x in range(0, 5):
            for s_y in range(0, 5):
                for b in np.linspace(0, 1, 4):
                    b = round(b, 1)
                    self.states.append((s_x, s_y, b))

        # action space
        self.actions = ((0,1),(0,-1),(1,0),(-1,0),(0,0))
        # discount factor
        self.gamma = 0.99
```

```
    # deterministic dynamics
    def f(self, s, a):

        # take commanded action
        next_states = {}
        s1 = self.take_action(s, a)

        # get human input
        # this occurs at a specific state
        # only at this state do we get insight from the
        # human about which goal to go for :)
        if s[0] == 4 and s[1] == 4:
            s1_goal1 = (s1[0], s1[1], 1.0)
            s1_goal2 = (s1[0], s1[1], 0.0)
            # half the time the human tells us theta_1
            # the other half the human tells us theta_2
            next_states[s1_goal1] = 0.3
            next_states[s1_goal2] = 0.7
        else:
            next_states[s1] = 1.0

        return next_states

    # helper function for dynamics
    def take_action(self, s, a):
        # s1 = s + a, belief stays constant
        s1 = (s[0] + a[0], s[1] + a[1], s[2])
        # keeps robot inside state space
        if s1 in self.states:
            return s1
        else:
            return s

    # reward for goal 1
    # this goal is at position (3, 1)
    def reward1(self, s):
        if s[0] == 3 and s[1] == 1:
            return +1.0
        else:
            return 0.0

    # reward for goal 2
    # this goal is at position (1, 3)
    def reward2(self, s):
        if s[0] == 1 and s[1] == 3:
            return +1.0
        else:
            return 0.0


# Q-function
def Qfunction(mdp, s, a, V):
    next_states = mdp.f(s, a)
    expected_V1 = sum(next_states[s1] * V[s1] for s1 in next_states)
    # get expected reward using belief
    expected_reward = s[2] * mdp.reward1(s) + (1.0 - s[2]) * mdp.reward2(s)
    return expected_reward + mdp.gamma * expected_V1

# get optimal policy from value function
def policy(mdp, V):
    pi = {}
    for s in mdp.states:
        # take action that maximizes Q-function
        max_Q = -np.inf
        for action in mdp.actions:
            Q = Qfunction(mdp, s, action, V)
            if Q > max_Q:
```

```
                max_Q = Q
                pi[s] = action
    return pi

# value iteration algorithm to get value function
def value_iteration(mdp, epsilon=0.001):
    V1 = {s: 0 for s in mdp.states}
    while True:
        V = V1.copy()
        delta = 0
        for s in mdp.states:
            # use value function V to get estimate V'
            action_values = []
            for action in mdp.actions:
                next_states = mdp.f(s, action)
                action_value = sum(next_states[s1] * V[s1] for s1 in next_states)
                action_values.append(action_value)
            # get expected reward using belief
            expected_reward = s[2] * mdp.reward1(s) + (1.0 - s[2]) * mdp.reward2(s)
            V1[s] = expected_reward + mdp.gamma * max(action_values)
            # find largest change in value
            delta = max(delta, abs(V1[s] - V[s]))
        # stop if largest change is small enough
        if delta <= epsilon * (1 - mdp.gamma) / mdp.gamma:
            return V

def main():

    # setup the pomdp
    # this functions the same as an mdp
    # but now with belief as part of state
    robot = POMDP()

    # use value iteration to get V
    V = value_iteration(robot)

    # get optimal policy from V
    pi = policy(robot, V)

    # print policy
    print(pi)

    # rollout policy from chosen start state
    s = (0, 0, 0.3)
    for t in range(20):
        a = pi[s]
        print(s, a)
        next_states = robot.f(s,a)
        # sorry this part is confusing
        # need to pick the next state
        s = list(next_states.keys())[0]

main()

```

```
{(0, 0, 0.0): (0, 1), (0, 0, 0.3): (0, 1), (0, 0, 0.7): (0, 1), (0, 0, 1.0): (0, 1), (0, 1, 0.0): (0, 1), (0, 1, 0.3):
(0, 0, 0.3) (0, 1)
(0, 1, 0.3) (0, 1)
(0, 2, 0.3) (0, 1)
(0, 3, 0.3) (1, 0)
(1, 3, 0.3) (0, 1)
(1, 4, 0.3) (1, 0)
(2, 4, 0.3) (1, 0)
(3, 4, 0.3) (1, 0)
(4, 4, 0.3) (0, -1)
(4, 3, 1.0) (0, -1)
(4, 2, 1.0) (0, -1)
(4, 1, 1.0) (-1, 0)
(3, 1, 1.0) (0, 0)
(3, 1, 1.0) (0, 0)
```

```
(3, 1, 1.0) (0, 0)
(3, 1, 1.0) (0, 0)
(3, 1, 1.0) (0, 0)
(3, 1, 1.0) (0, 0)
(3, 1, 1.0) (0, 0)
(3, 1, 1.0) (0, 0)
```

3. Latent Actions

3.1

Imagine that you have a matrix of 7-dimensional robot actions. You want to control the robot using 2-DoF. You propose finding these embeddings through singular value decomposition. Write detailed pseudocode for controlling the robot using the first two eigenvectors.

Assuming I have 2 DoF, this means a have only 2 axis, in this case I am going to say X,Y.

X+ -> Output_1: When we move the X vector to positive direction we control the Output 1

X- -> Output_2: When we move the X vector to negative direction we control the Output 2

Y+ -> Output_3: When we move the Y vector to positive direction we control the Output 3

Y- -> Output_4: When we move the X vector to negative direction we control the Output 4

Sin(citha) -> Output_5: The Sin of the angle citha, that is the angle beetwen X and Y, control the Output 5

Cos(citha) -> Output_6: The Cos of the angle citha, that is the angle beetwen X and Y, control the Output 6

Tan(citha) -> Output_7: The Tan of the angle citha, that is the angle beetwen X and Y, control the Output 7

3.2 Now consider the opposite direction; you have a 7-dimensional input device and are controlling a 2-DoF robot. Describe your own algorithm for mapping high-dimensional inputs to low-dimensional outputs. Explain why your approach would be intuitive for the user.

Assuming that I have 7 input device, like a control with 4 arrows and 3 buttons, to control 2 DoF (X,Y), I would do this:

Map the Input 1 to control X positive direction with the current citha angle that is the angle beetwen X and Y

Map the Input 2 to control X negative direction with the current citha angle that is the angle beetwen X and Y

Map the Input 3 to control Y positive with the current citha angle that is the angle beetwen X and Y

Map the Input 4 to control tranformsY negative direction with the current citha angle that is the angle beetwen X and Y

Map the Input 5 to transform the Input 1, 2, 3 and 4 to control the citha angle that is the angle beetwen X and Y from 0 to 180

Map the Input 6 to transform the Input 1, 2, 3 and 4 to control the citha angle that is the angle beetwen X and Y from 180 to 360

Map the Input 7 to activate the angle movement (5-6 become usable) or only use the 1D movement of Inputs 1,2,3 and 4

In this case would be intituive because with the arrows they can move normaly the robot, and with the 4 and 5 buttons they can activate the funtion to convert the arrows into mappings of 1D to Angle vectors, and with the other button they can activate or disactivate this mapping angle functions. With this approch the arrows will be only managing a specif task depending of the combination or 5, 6 and 7

✓ 8 s se ejecutó 17:04