# 1

The task environment of a soccer robot can be described as **partially observable, stochastic, sequential, dynamic, continuous**, and **multi-agent**.

* **[Partially observable]** as the robot cannot fully observe the entire view of the soccer playground, such as the positions and actions of other players.
* **[Stochastic]** as the actions of other players are influenced by various unpredictable factors, such as their movements and their decisions.
* **[Sequential]** as the robot's actions affect the environment's subsequent state, the robot must make decisions based on past experiences.
* **[Dynamic]** as the environment is constantly changing due to the movements of the players and the ball.
* **[Continuous]** as the environment can have an infinite number of states and the possibilities of taking an action are also infinite since the robot can take a wide range of possible actions, such as running, kicking, or passing the ball.
* **[Multi-agent]** as the robot, interacts with other players on both its own team and the opposing team and must make decisions based on the actions of all players on the field.

# 2

Graphical user interface

Description automatically generated

import PIL.Image as Image  
import numpy as np  
import matplotlib.pyplot as plt  
  
def guassian\_filter(img, sigma=1):  
 # get size of image  
 m, n = img.shape  
 # get size of kernel  
 size = int(6 \* sigma + 1)  
 # get center of kernel  
 center = size // 2  
 # initialize kernel  
 kernel = np.zeros((size, size), dtype=np.float32)  
 # calculate kernel  
 for i in range(size):  
 for j in range(size):  
 x, y = i - center, j - center  
 kernel[i, j] = np.exp(-(x\*\*2 + y\*\*2) / (2 \* sigma\*\*2))  
 kernel /= (2 \* np.pi \* sigma\*\*2)  
 # initialize output image  
 output = np.zeros((m, n), dtype=np.float32)  
 # convolution  
 for i in range(m):  
 for j in range(n):  
 for k in range(size):  
 for l in range(size):  
 ii = i + k - center  
 jj = j + l - center  
 if ii >= 0 and ii < m and jj >= 0 and jj < n:  
 output[i, j] += img[ii, jj] \* kernel[k, l]  
 return output  
  
def sobel\_filter(img):  
 # get size of image  
 m, n = img.shape  
 # initialize output image  
 output = np.zeros((m, n), dtype=np.float32)  
 # convolution  
 for i in range(1, m - 1):  
 for j in range(1, n - 1):  
 output[i, j] = np.sqrt((img[i - 1, j - 1] + 2 \* img[i - 1, j] + img[i - 1, j + 1] - img[i + 1, j - 1] - 2 \* img[i + 1, j] - img[i + 1, j + 1])\*\*2 +  
 (img[i - 1, j - 1] + 2 \* img[i, j - 1] + img[i + 1, j - 1] - img[i - 1, j + 1] - 2 \* img[i, j + 1] - img[i + 1, j + 1])\*\*2)  
 return output  
  
def non\_max\_suppression(img):  
 # get size of image  
 m, n = img.shape  
 # initialize output image  
 output = np.zeros((m, n), dtype=np.float32)  
 # non-max suppression  
 for i in range(1, m - 1):  
 for j in range(1, n - 1):  
 # get gradient direction  
 direction = np.arctan2(img[i, j], img[i, j])  
 # get pixel value of 2 neighbors  
 p1, p2 = 255, 255  
 if (0 <= direction < np.pi / 8) or (7 \* np.pi / 8 <= direction <= np.pi):  
 p1, p2 = img[i, j + 1], img[i, j - 1]  
 elif np.pi / 8 <= direction < 3 \* np.pi / 8:  
 p1, p2 = img[i + 1, j - 1], img[i - 1, j + 1]  
 elif 3 \* np.pi / 8 <= direction < 5 \* np.pi / 8:  
 p1, p2 = img[i + 1, j], img[i - 1, j]  
 elif 5 \* np.pi / 8 <= direction < 7 \* np.pi / 8:  
 p1, p2 = img[i + 1, j + 1], img[i - 1, j - 1]  
 # compare pixel value  
 if img[i, j] >= p1 and img[i, j] >= p2:  
 output[i, j] = img[i, j]  
 else:  
 output[i, j] = 0  
 return output  
  
def threshold(img, low\_threshold\_ratio=0.05, high\_threshold\_ratio=0.09):  
 # calculate high and low threshold  
 high\_threshold = img.max() \* high\_threshold\_ratio  
 low\_threshold = high\_threshold \* low\_threshold\_ratio  
 # get size of image  
 m, n = img.shape  
 # initialize output image  
 res = np.zeros((m, n), dtype=np.float32)  
 # thresholding  
 weak = np.int32(25)  
 strong = np.int32(255)  
 strong\_i, strong\_j = np.where(img >= high\_threshold)  
 zeros\_i, zeros\_j = np.where(img < low\_threshold)  
 weak\_i, weak\_j = np.where((img <= high\_threshold) & (img >= low\_threshold))  
 res[strong\_i, strong\_j] = strong  
 res[weak\_i, weak\_j] = weak  
 return (res, weak, strong)  
  
def hysteresis(img, weak, strong):  
 # get size of image  
 m, n = img.shape  
 for i in range(1, m-1):  
 for j in range(1, n-1):  
 if img[i, j] == weak:  
 try:  
 if ((img[i + 1, j - 1] == strong) or (img[i + 1, j] == strong) or (img[i + 1, j + 1] == strong) or  
 (img[i, j - 1] == strong) or (img[i, j + 1] == strong) or  
 (img[i - 1, j - 1] == strong) or (img[i - 1, j] == strong) or (img[i - 1, j + 1] == strong)):  
 img[i, j] = strong  
 else:  
 img[i, j] = 0  
 except IndexError as e:  
 pass  
 return img  
  
def canny\_edge\_detection(img, sigma):  
 # guassian\_filter  
 img = guassian\_filter(img, sigma)  
 # sobel\_filter  
 img = sobel\_filter(img)  
 # non-max suppression  
 img = non\_max\_suppression(img)  
 # thresholding  
 img, weak, strong = threshold(img)  
 # hysteresis  
 img = hysteresis(img, weak, strong)  
 return img  
  
def plot(img, canny):  
 # plot original image and canny edge detection  
 plt.subplot(121)  
 plt.imshow(img, cmap='gray')  
 plt.title('Original Image')  
 plt.axis('off')  
 plt.subplot(122)  
 plt.imshow(canny, cmap='gray')  
 plt.title('Canny Edge Detection')  
 plt.axis('off')  
 plt.show()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 # read image  
 img = Image.open('img.jpg')  
 # img to grayscale  
 img = img.convert('L')  
 # img to array  
 img = np.array(img)  
 # show original image and canny edge detection  
 plot(img, canny\_edge\_detection(img, sigma=1))