Sebastian Bliefert
Nils Drebing
Pascal Pieper

Dozent: Marc Otto
Gruppe: G02
Abgabedatum: 8.12.2016

Verhaltensbasierte Robotik (WiSe 16/17)

Lösungsvorschlag zu Übungsblatt 2:

Inhaltsverzeichnis

a). abc

Aufgabe 1

Interne Sensoren messen den Zustand der Roboters (z.B. Wärmesensoren in Gelenken), externe Sensoren dagegen den Zustand der Welt um den Roboter (z.B. Laserscanner). Dabei kann dann noch unterschieden werden zwischen aktiven und passiven Sensoren. Passive messen nur die Welt um sich herum (oder auch den Zustand des Roboters falls es ein interner Sensor ist) (z.B. eine Kamera). Aktive Sensoren manipulieren die Welt (oder was auch immer sie messen wollen) zusätlich noch wie zum Beispiel Laserscanner, die einen Laserstrahl aussenden um Entfernungen zu messen.

Wärmesensoren funktionieren mittels eines Leiters (Heiß- oder Kaltleiter) der bestromt wird. Dadurch, dass sich der Widerstand des Leiters mit der Temperatur des Leiters ändert kann an der resultierenden Stromstärke die Temperatur des Leiters ermittelt werden. Dies geschieht dann häufig softwareseitig, wo die Stromstärke in die Temperatur umgerechnet wird.

Laserscanner senden einen Laserstrahl in eine Richtung aus und messen, wie lange es dauert bis dieser zurück kommt. Anhand dieser Messung kann mit Hilfe der Kenntnis über die Lichtgeschwindigkeit die Entfernung zu dem Punkt, von dem er reflektiert wurde ermittelt werden. Dies Passiert nun in hoher geschwindigkeit hintereinander und in verschiedene Richtungen, wodurch eine Punktwolke generiert werden kann, die die Welt um den Roboter abbildet.

Aufgabe 2

Point Turn

Mit dieser Methode kann der Roboter sich auf der Stelle drehen. Die Geschwindigkeit wird aus einer globalen Variable *speed* gezogen.

```
def doPointturn():
    global right_actuator
    global left_actuator
    right_actuator = speed/2.
    left_actuator = -speed/2.
```

Kamerabildalgorithmus

Zur Erkennung des roten Balls wird nach roten Pixeln mit entsprechenden RGB-Werten gesucht. Dabei wurden die Parameter durch *trial and error* ermittelt. Der Algorithmus

endet, sobald ein einziges rotes Pixel im Bild gefunden wurde. Als Rückgabe wird ein Tupel zurückgegeben, das zum einen angibt, in welchem Kamerabild das Pixel gefunden wurde. Zum anderen wird die vertikale Position des Pixels im Bild zurückgegeben, wobei die Nullkoordinate in der Mitte des Bildes liegt.

```
def trackBall():
    global lcam, rcam
    global minX,maxX
    redPix = 0
    red = False
    for i in range(len(lcam)):
        if lcam[i][0] >= 200. and lcam[i][1] <= 100. and lcam[i][2] <= 100.:
            #move to zero as center
            approx_vert = i % width - width/2.
            if(approx_vert < minX):
                minX = approx_vert
            if(approx_vert > maxX):
                maxX = approx_vert
            red = True
    if(red):
            return ("left", approx_vert)
    for i in range(len(rcam)):
        if rcam[i][0] >= 200. and rcam[i][1] <= 100. and rcam[i][2] <= 100.:
            approx_vert = i % width - width/2.
            if(approx_vert < minX):
                minX = approx_vert
            if(approx_vert > maxX):
                maxX = approx_vert
            red = True
    if(red):
            return ("right", approx_vert)
    return ("none", -1)
```

Je nach dem, wo sich der Ball im Bild befindet (tendenziell links, tendenziell rechts), wird der Motorwert gewichtet angepasst. Auf diese Weise wird der Ball in der Mitte der rechten Kamera gehalten.

Ballfindung

```
def approach(position):
    global right_actuator
    global left_actuator
    if(position[0] == "left"):
        right_actuator = speed
        left_actuator = speed/2
    if(position[0] == "right"):
        right_actuator = speed
        left_actuator = speed + float(position[1]) / float(width)*speed
```

Kugel umrunden

Hier soll die Kugel im Zentrum des Bildes der linken Kamera behalten. Die Motorwerte werden je nach Position des Balls im Bild angepasst, sodass eine kreisförmige Trajektorie abgefahren wird.

```
def circle():
    global left_actuator, right_actuator
    global minX, maxX
    global state
    logMessage("Circling around object!")

    bawl = trackBall()

    left_actuator = 1.
    right_actuator = 0.

    if(bawl[0] == "left"):
        factor = -.15   #'magic'
        factor += (bawl[1] / (width / 2.)) * speed
        #logMessage("Factor: " + str(factor))
        if(diameter(minX, maxX) < diameterTh):
            logMessage("Too far away, turning inside a bit")
            factor -= 1.

        left_actuator  = speed / 2. + factor
        right_actuator = speed / 2. - factor
    return True

    if(bawl[0] == "none"):
        return False
```

Kugeln an die Wand

Nicht implementiert.

Statemachine

```
def execute(camData):
    global state
    global t
    global right_actuator, left_actuator
    global maxX,minX
    minX = 10000
    maxX = -10000

    readData(camData)
```

```
if(state == "pt"):
    doPointturn()
    if(checkForRed() == True):
        logMessage("switching to state approach, checkforred is true")
        state = "approach"
        return
if(state == "approach"):
    position = trackBall()
    #print(diameter(minX, maxX))
    if(diameter(minX, maxX) >= diameterTh):
        logMessage("now too close! switching to state circle")
        state = "circle"
        return
    if(position[0] != "none"):
        approach(position)
    else:
        logMessage("switching to state pt, position equals none")
        state = "pt"
        return
if(state == "circle"):
    stillCircling = circle()
    if(!stillCircling):
        state = "pt"
```

Aufgabe 3

Unsere Architektur benutzt eine Statemachine für die Verwaltung der Zustände. Die Kugel wird gefunden, indem der Roboter sich dreht, bis er eine rote Kugel (bzw. Rote Pixel) sieht. Dann dreht er sich (schon leicht fahrend) zu der Kugel, bis sie sich in der Mitte seiner Frontkamera befindet. Dann fährt er geradeaus. Beim Fahren korrigiert er seine Richtung leicht in Abhängigkeit der Verschiebung der Kugel. Wenn die Kugel groß genug ist, wechselt er in den Umrundungsmodus. Dort dreht er sich, bis die Kugel in seiner Seitenkamera auftaucht. Dann fährt er geradeaus und dreht sich gleichzeitig in Abhängigkeit der Position der Kugel. Also fährt der Roboter geradeaus, wenn die Kugel in der Mitte seiner Seitenkamera ist; dreht anteilig links, wenn die Kugel links verschoben ist, und anders herum. Wenn die Kugel aus dem Sichtfeld verschwindet, geht er wieder in den Findungszustand.
Diese Zustände sind unabhängig voneinander. Alle Verhalten sind in sich geschlossen. States können durch eine *if*-Klausel in der *execute()*-Funktion hinzugefügt werden.
Die Änderung der Zustände können in jedem Durchlauf einer Bildverarbeitung (`execute()`) gemacht werden. Je mehr Kamerabilder pro Sekunde berechnet werden, desto schneller kann der Roboter um die Kugel kreisen. blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabvalbalbalba

blkabval

b). abcd