Robotic Games Sommersemester 2019
Eine Mauß in 150 Zeilen Code

Tobias Bak

17. August 2019

1 Problemstellung

Dieses Dokument enthält einen Nachtrag zur Implementierung des Katze und Maus Spiels mit dem Pioneer 3 Roboter.[3]

Als Aufgabe wurde an eine Gruppe von vier Personen die Anforderung den Pioneer Roboter so zu Programmieren dass dieser autonom die Mausrolle eines Katz und Maus spiels zu realisieren. Meine Gruppe hatte die Aufgabe sehr einfach gelöst ohne Verhaltensfusion und ohne reaktiven Verhalten auf die Umgebung. Das Bedeutet der Roboter der unsere Gruppe Programmiert hatte konnte nur hinter einen Vorhang raus und zum Ziel fahren. Die Einfache Implementierung hatte großen Vorteil bei dem Spiel und gewann dieses auch mit Abstand jedoch wurde die Aufgabe nicht in der richtigen Weise gelöst. Aus diesem Grund wurde eine neue Implementierung von Grund auf aufgebaut. Ziel war es hier auch sich die Randbedingung des Spiels zu nutze zu machen um das Verfahren möglichst elegant und simple zu halten um einerseits Rechenleistung zu sparen und das gesammte System so zu beschleunigen und andererseits eine lange Ausarbeitung zu umgehen, da die hier vorliegende Arbeit nur von mir geschrieben wurde.

Abbildung 1: Pioneer 3 [3]

Die hier angenommenen Randbedingungen sind dabei die einer bis auf Verstecke konvexer Arena, einer Drehgeschwindigkeit die deutlich größer ist als die Drehgeschwindikgeit sowie bekannter Maße und Farben der Hindernisse, Käsestücke und Katze.

Es war somit möglich das gesammte System in 150 Zeilen zu implementieren aufgeteilt in eine Launch File, ein Skript zur extrahierung optischer Daten, eines zur extrahierung von Sonarbasierten daten sowie der Hauptcode welcher das intelligente Verhalten realisiert. Es wird hier zunächst mit diesem letzten Teil begonnen, da die Form der in den anderen Programmen generierten Daten auf eben dieses Programm angepasst ist.

2 Fusion Node

Zunächst soll mit einem Blick auf den Code begonnen werden, welcher hier im folgenden dargestellt ist, der Wichtige Teil ist dabei in Zeilen 7-51 zu sehen. Im folgenden wird der Code nun durchgegangen und einzelne wichtige Stellen erläutert.

```
1  #!/usr/bin/env python
2  import sys
3  import rospy
4  import numpy as np
5  from geometry_msgs.msg import Twist
6  from geometry_msgs.msg import Point
7  class Fusion:
8          def __init__(self):
9                  # array containes cat,blue,orange,cheese,collision
10                 self.behaviors = np.zeros((5,2))
11                 parameters     =  np.array([[-0.5,-1,-1],[0.3,0,1],[0,0,1],[0.8,0,1],[-0.5,-1,-1]])
12                 for i in range(5):
13                     rospy.Subscriber(sys.argv[i+1], Point, self.generate_behavior,
14                     (i,parameters[i,0],parameters[i,1],parameters[i,2]))
15                 self.pub                    = rospy.Publisher('/RosAria/cmd_vel', Twist, queue_size=1)
16                 while not rospy.is_shutdown():
17                         self.fusion()
18         def generate_behavior(self,msg,args):
19                 print(args[0])
20                 if msg.x != 0 and msg.y != 0 :
21                     self.behaviors[args[0]] = np.array([1,args[1]*msg.x**args[2]*msg.y**args[3]])
22                 else:
23                     self.behaviors[args[0]] = np.array([1,0])
24
25         def fusion(self):
26                 #print('collision', self.collision, 'cheese', self.cheese)
27                 cmd_vel = Twist()
28                 cmd_vel.linear.x = 1
29                 cmd_vel.angular.z = prevail_gate(or_gate(
30                 invoke_gate(self.behaviors[2,1],self.behaviors[0,1]),self.behaviors[3,1]),self.behaviors[4,1])
31                 #cmd_vel.angular.z = self.behaviors[1,1]
32                 print(self.behaviors)
33                 self.pub.publish(cmd_vel)
34 def and_gate(x, y):
35         a = 2.28466
36         b = -0.89817
37         if x == 0 and y == 0:
38             return 0
39         else:
40             return x*(1-np.exp(-((a*y**2+b*x*y)/(x**2+y**2))))+ y*(np.exp(-((a*x**2+b*x*y)/(x**2+y**2))))
41 def or_gate(x, y):
42         a = 1.02889
43         b = 0.3574
44         if x == 0 and y == 0:
45             return 0
46         else:
47             return x*(np.exp(-((a*y**2+b*x*y)/(x**2+y**2)))) + y*(np.exp(-((a*x**2+b*x*y)/(x**2+y**2))))
48 def invoke_gate(x, y):
49         return and_gate(or_gate(x, y),x)
50 def prevail_gate(x, y):
51         return or_gate(x, or_gate(x, y))
52 if __name__ == '__main__':
53         try:
54                 rospy.init_node("fusion")
55                 fus = Fusion()
56         except rospy.ROSInterruptException:
57                 rospy.loginfo("---------- FUSION-ERROR! ----------")
```

Bei initilalisierung des Programms werden zunächst einzelne Speicher für jedes Verhalten des Roboters zu gewiesen, die hier verwendeten Verhalten sind dabei Verstecken, Fliehen, Käse einsammeln und Kollisionsvermeidung. Die Einzelnen Verhalten selbst werden dabei im Callback des Subscribers der jeweiligen Messdaten erzeugt.

Dabei wird davon Ausgegangen das die eingehenden Daten die Form einer Position relativ zum Roboter haben. abstrakt kann man nun die Verhalten in 2 Klassen aufteilen: Das Regeln dieser Position auf den Sollwert Null und das Regeln auf den Sollwert Unendlich.

Aufgrund der langsamen lineargeschwindigkeit des Roboters ist es dabei volkommen ausreichend nur die Drehgeschwindigkeit zu regeln und die lineargeschwindigkeit konstant zu lassen, auch wenn wie im Code zu sehen ist, ein Verhalten immer aus 2 Größen besteht um den Code im Zweifel zu erweitern. Beide Klassen von Verhalten lassen sich nun mit der

`generate_behavior` Funktion in Zeilen 18-23 realisieren. Diese verwendet eine reihe von Inputs welche für jedes Verhalten in Zeile 11 Definiert sind. Mathematisch implementiert die Funktion die folgende Gleichung für die ausgegebene Geschwindigkeit v:

$$v = p_1 * \rho^{p_2} * \phi^{p_3} \tag{1}$$

Dabei sind ρ und ϕ der Abstand und Winkel zum Roboter und seiner Fahrtrichtung. für Verhalten welche die Position zu Null regeln wollen führt die Parameter Konfiguration (k,0,1) zu einem Proportionalregler welcher die Orientierung des Roboters so regelt dass er exakt auf das Ziel zu fährt. Dies ist wie in 11 zu sehen sowohl für die Verstecke als auch den Käse der Fall. Für das Regeln auf Unendlich wollen wir uns möglichst stark von unserem Ziel wegdrehen, die Drehung sollte dabei stärker sein wenn das Ziel im Zentrum und oder sehr nahe ist, mathematisch erhält man damit die folgende Form:

$$v = \frac{p_1}{\rho^{p_2} * \phi^{p_3}} \qquad p_2, p_3 \text{ungerade} \tag{2}$$

Das Regeln zu unendlich kann also mit einer Art proportional Regler mit umgekehrten Vorzeichen in allen Parametern implementiert werden.

Wurden die einzelnen Verhalten generiert so werden sie in Zeile 28-36 Fusioniert, diese Fusion folgt gates und verwendet die in Zeile 37-54 definierten Analogen Gatter

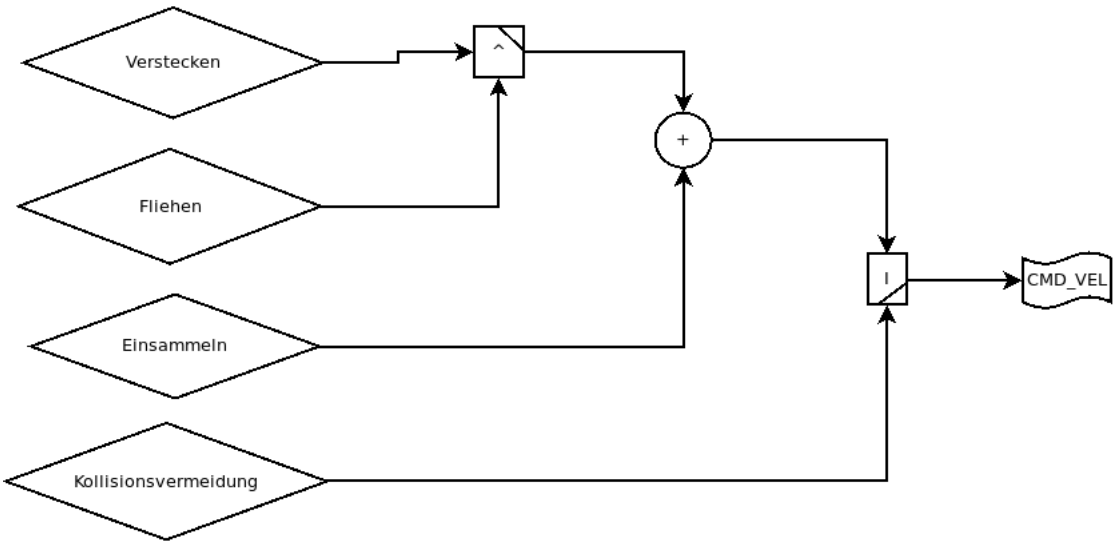


Abbildung 2: Analogical Gates

Anschaulich kann das Verhalten wie folgt verstanden werden:
Die Maus versucht zunächst in gleichen Teilen der Katze auszuweichen und den Käse zu erhalten, dies wird über ein analoges OR realisiert was der reinen Winkelgeschwindigkeitsregelung geschuldet ist und dafür sorgt dass sich die Maus nicht immer sofort von der Katze wegdreht, sondern einen großen Bogen um sie fährt.

Verstecken wiederum macht nur Sinn wenn man vor der Katze fliehen will und wird entsprechend stärker je stärker der Roboter vor der Katze fliehen will. Dies wird durch ein INVOKE Gatter realisiert. Kollisionen sollten selbstverständlich zu jedem Zeitpunkt

vermieden werden und so überschreibt die Kollisionsvermeidung jedes andere Verhalten mittels einens PREVAIL Gatters.

Nun da man verstanden hat, wie der Roboter opperiert muss betrachtet werden, wie er die Positionsdaten für die einzelnen Verhalten erhält, hier soll dabei mit den Sonardaten begonnen werden.

3 Ultraschall

Das Ultraschallmodul ist zunächst für das Lesen der Ultraschallsensoren zuständig. Aus gelesenen Ultraschalldaten wird eine Position eines zu vermeidendes Objekts für die Verhaltensfusion veröffentlicht. Zusätzlich wird in diesen Modul die Kollisionsvermeidung für das Versteckt ausgeschaltet. Das Modul bindet also nur die Ultraschallsensoren in das Programm ein und berechnet in welchen Winkel sich die kleinste gelesene Distanz befindet und gibt die Entfernung und den Winkel aus.

Um dies zu ermöglichen benötigt die Node ebenfalls die Entfernungen zu den einzelnen Verstecken. Die Tatsächliche berechnung passiert im `sonar_callback` in Zeilen 80-92. Zuerst wird $d_o(\alpha)$ berechnet. $d_o(\alpha)$ ist dabei die berechnete Distanz vom Sensor der mit dem Winkel α am Roboter ausgerichtet ist. Die kollisionsvermeidung soll nun unterdrückt werden, wenn sich der Roboter nahe an einem Vorhang befindet, da das Ausweichverhalten mit dem Abstand skaliert, wird dies durch eine modifizierung von $d_o(\alpha)$ zu $d_i(\alpha)$ verwirklicht. Die Idee ist also den Abstand beliebig groß zu stellen, wenn das Versteck beliebig nahe ist, da das entstehende Verhalten so beliebig Schwach wird. Sei d_v die Vektoren welche die Entfernungen zu beiden Vorhängen beschreibt, dann sind die Ausgabe Distanzen wie folgt definiert (Zeilen 85,86):

$$d_o(\alpha) = \sqrt{d_i(\alpha)_x^2 + d_i(\alpha)_y^2} \cdot 2 \cdot (\frac{1}{1 + e^{-min(d_v)}} - 0,5) \tag{3}$$

Anschließend wird der Kleinste Abstand in d_o gesucht und mit dem zugehörigen Winkel zusammen ausgegeben abschließend durch die Verhaltensfusion weiter verarbeitet(Zeilen 87-91). Der Code hierfür sieht wie folgt aus:

```
58 #!/usr/bin/env python
59 import numpy as np
60 import rospy
61 from sensor_msgs.msg import PointCloud
62 from geometry_msgs.msg import Point
63 max_distance = 1
64 weight       = np.array([  0.5,   0.7,   1.0,   1.0,  1.0,  1.0,  0.7,  0.5])
65 sonar_angles = np.array(
66 [-90.0, -50.0, -30.0, -10.0, 10.0, 30.0, 50.0, 90.0])/ 360.0 * 2 * np.pi
67 class Sonar:
68     def __init__(self):
69         self.hideout = np.zeros(2)
70         obst_sub  = rospy.Subscriber("/RosAria/sonar",PointCloud,self.sonar_callback)
71         blue_sub  = rospy.Subscriber("blue_position",Point,self.hideout_callback,0)
72         orang_sub = rospy.Subscriber("orange_position",Point,self.hideout_callback,1)
73         while not rospy.is_shutdown():
74             print("still working")
75     def hideout_callback(self,data,i):
76         if data.x != 0:
77             self.hideout[i] = data.x
78         else:
79             self.hideout[i] = 100000
80     def sonar_callback(self,data):
81         pub       = rospy.Publisher("obstacle_position",Point,queue_size=1)
82         sonar_points = data.points
83         sonar_ranges = np.zeros(len(sonar_angles))
84         for i in range(0, len(sonar_angles)):
```

```
85              sonar_ranges[i]=np.sqrt(sonar_points[i].x**2+sonar_points[i].y**2)*2*
86              (1/(1+np.exp(-1*(min(self.hideout))))-0.5)
87          minimum  = np.argmin(sonar_ranges)
88          output   = Point()
89          if sonar_ranges[minimum] <= max_distance:
90              output.x = sonar_ranges[minimum]*weight[minimum]
91              output.y = -sonar_angles[minimum]
92          pub.publish(output)
93  if __name__== '__main__':
94      rospy.init_node("obstacle_detection")
95      try:
96          node=Sonar()
97      except rospy.ROSInterruptException:
98          rospy.loginfo("sonar not working")
```

4 Vision

Die Vision ist das Modul um Objekte per Kamera zu erkennen. Es wurde während dieses Projekts nach Farben selektiert, so waren die Katzen Rot, die Vorhänge Gelb und Blau und der Käse Grün. Das Vision Modul sucht abhängig von gewählten Farbe nach Objekten dieser Farbe und Gibt die Entfernung und den Winkel des Objekts aus welchen von der Verhaltensfusion weiter verarbeitet werden.

Im Ablauf ist das Modul wie folgt aufgebaut. Zunächst wird das Bild in den HSV Farbraum konvertiert (Zeile 117) anschließend werden über das Bild zwei Masken gelegt. Eine Maske schneidet den obere Bildhälfte ab (Zeile 120) die andere Maske blendet alle Farben die in nicht definierten Raum befinden aus (Zeile 118,119). So vorbereitetes Bild wird nach Konturen durchgesucht. In den Konturen selbst wird die größte Kontur gesucht und eine Box um diese gelegt. Aus den Maßen der Box werden anschließend der Winkel α und die Entfernung d berechnet (Zeilen 129-132). Wobei f die Fokuslänge, x, y die Position des Pixels am weitesten links oben und w, h die Breite und die Höhe des Objekts sind sowie img_w die Gesamtbreite des Bildes ist. Damit ergibt sich für den Winkel α:

$$\alpha = -\arctan(\frac{(x + w \cdot 0,5 - \frac{img_w}{2})}{f}) \tag{4}$$

und für die Entfernung:

$$d = \frac{f}{h * \cos(\alpha)} \tag{5}$$

Zusätzlich ist es noch möglich in Abwesenheit eines Objekts den letzten Winkel durchzugeben, das bewirkt das der Roboter bei der Flucht vor der Katze in der Lage ist sich wieder zurück zum Käse zu drehen (Zeilen 122-124). Der Code hierfür sieht wie folgt aus:

```
99  #!/usr/bin/env python
100 import sys
101 import cv2
102 import rospy
103 from sensor_msgs.msg import Image
104 from geometry_msgs.msg import Point
105 from cv_bridge import CvBridge
106 import numpy as np
107 focal_len        = 500
108 min_size         = 600
109 max_height       = 200
110 global last_cheese
111 last_cheese      = 0
112 def img_call(ros_img):
113         global last_cheese
114         cv_img                  = bridge.imgmsg_to_cv2(ros_img,'rgb8')
115         image_h, image_w        = cv_img.shape[:2]
116         hsv_img                 = cv2.cvtColor(cv_img, cv2.COLOR_RGB2HSV)
117         mask                    = cv2.inRange(hsv_img,np.array([int(sys.argv[3]),int(sys.argv[4]),
118         int(sys.argv[5])]) , np.array([int(sys.argv[6]),int(sys.argv[7]),int(sys.argv[8])]) )
119         mask                    = cv2.rectangle(mask,(0,0),(len(mask[0]),max_height),(0,100,0),-1)
120         im2, contours, hierarchy = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
121         output                  = Point()
```

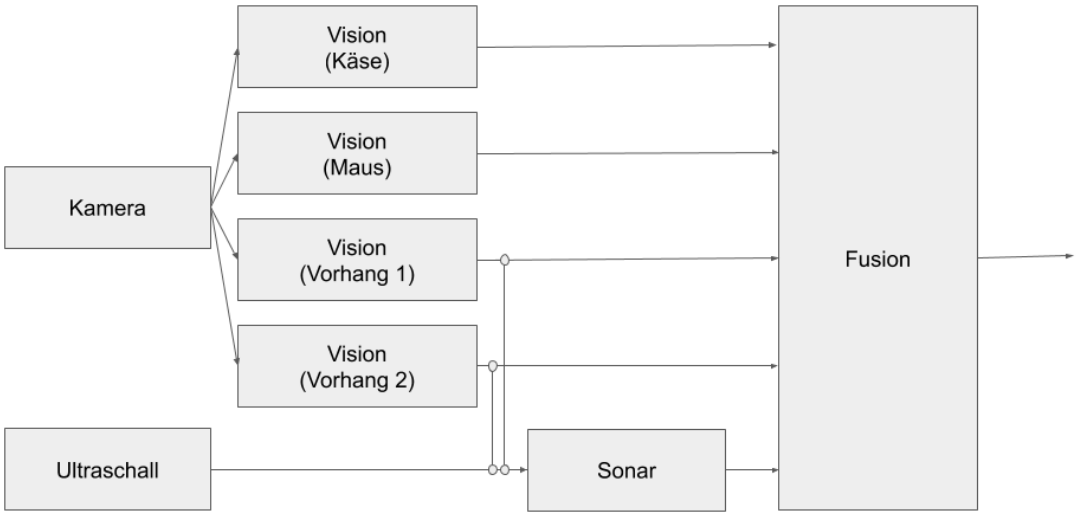
```
122             if int(sys.argv[9]) == 1:
123                 output.x                    = 1
124                 output.y                    = last_cheese
125             if len(contours) > 0:
126                     biggest_contour=max(contours, key = cv2.contourArea)
127                     if cv2.contourArea(biggest_contour) >= min_size:
128                             x,y,w,h         = cv2.boundingRect(biggest_contour[0])
129                             output.y            = -1*np.arctan((x+w*0.5-image_w/2.0)/focal_len)
130                             output.x            = float(sys.argv[2])*focal_len/(h*np.cos(output.y))
131                             last_cheese         = output.y
132             pubCheese.publish(output)
133 if __name__ == '__main__':
134             try:
135                     rospy.init_node("vision")
136                     bridge          = CvBridge()
137                     cam_sub         = rospy.Subscriber('/usb_cam/image_raw',Image,img_call)
138                     pubCheese       = rospy.Publisher(sys.argv[1], Point, queue_size=1)
139                     rospy.spin()
140             except rospy.ROSInterruptException:
141                     rospy.loginfo("vision type node not working")
```

5 Struktur

zum schluss muss noch auf die Programmstruktur eingegangen werden, wie in Zeilen 13,118,119,139 zu sehen ist, sind viele Parameter wie Subscriber namen und HSV werte im Code selbst nicht gesetzt, das liegt daran, dass die Vision Node mehrfach mit anderen Parametern initialisiert werden muss um alle Objekte zu erkennen, dies wird mit einer sogenannten Launch File implementiert, die in unserem Fall die Porgrammstruktur vorgibt und es ermöglicht Flexbile änderungen am Programm und seinem Fluss vorzunehmen, sowie den Roboter mit einem einzigen Befehl starten zu können:

```
142 <launch>
143 <node name="cheese_detection" type="vision.py" pkg="maus" args="cheese_position .1 60 60 0 80 255 100 1"/>
144 <node name="cat_detection" type="vision.py" pkg="maus" args="cat_position .18 0 120 30 22 255 125 0"/>
145 <node name="blue_hideout_detection" type="vision.py" pkg="maus" args="blue_position .66 80 30 100 120 255 255 0"/>
146 <node name="orange_hideout_detection" type="vision.py" pkg="maus" args="orange_position .66 20 50 120 25 120 255 0"/>
147 <node name="obstacle_detection" type="sonar.py" pkg="maus" />
148 <node name="fusion" type="fusion.py" pkg="maus" args="cat_position blue_position orange_position cheese_position
149                                                     obstacle_position /RosAria/cmd_vel" />
150 </launch>
```

Die Endgültige Struktur hat nun die folgende Form



Literatur

[1] A. Alexopoulos, T. Schmidt, and E. Badreddin. Pursuit and evasion in a recursive nested behavioral control structure for unmanned aerial vehicles. In *2014 14th International Conference on Control, Automation and Systems (ICCAS 2014)*, pages 1175–1180, Oct 2014.

[2] E. Badreddin. Analogical gates: A network approach to fuzzy control with applications to a non-holonomic autonomous mobile robot. *Intelligent Automation & Soft Computing*, 4(1):3–18, 1998.

[3] M. R. Inc. Pioneer 3 operations manual with mobilerobots exclusive advanced robot control and operations software. 2006.