Universität Bonn

Projektgruppe mobile Robotik WS 2016/17

Dr. Nils Goerke

Nessa Aananu

Marc Goedecke

Fabian Kahl

Inhaltsverzeichnis

1. Einführung

Meine Projektgruppe absolvierte ich im Blockkurs im Wintersemester 16/17. Dabei lernten wir den Staubsaugerroboter Roomba kennen. Dieser besitzt einen Laserscanner der Marke Sick der die Entfernung in m als Array published. Die Laserscanner, wie in Abb. 1.1 dargestellt gibt die Laserdaten als Lasernummern gegen den Uhrzeigersinn zurück. Er ließt die Werte in 0.5° Schritten ein und hat damit bei einer Sichtweite von 270°, #540 Lasernummern. Bei #270 wird also die Entfernung entlang der eigenen X-Achse angegeben. Die Drehgeschwindigkeit wird in Bogenmaß ausgegeben, dazu müssen zunächst die Lasernummern mit 0.5° (Schritte in den der Laserscanner misst) multipliziert werden und man erhält das Gradmaß. Um nun vom Gradmaß ins Bogenmaß umrechnen zu können ist es wichtig zu wissen, dass 360° im Gradmaß genau 2π im Bogenmaß entsprechen. Weiterhin benötigen wir folgende Formeln:

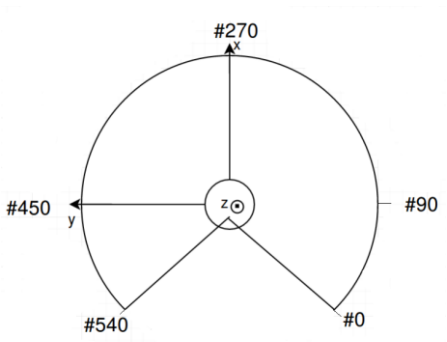


Abb. 1.1

Bogenmaß = (2*π/360°)*α°

α° = (360°/ 2*π) * Bogenmaß

Dem Roomba wurde für die wissenschaftliche Arbeit, die Saugeinheit entfernt. Weiterhin ist ein Notebook auf ihnen angebracht, welches zur Steuerung dient. Der Laserscanner ist über einen Anschluss mit dem Roomba verbunden und verbraucht sehr viel Akku daher sollte er nicht lange angeschlossen sein. Außerdem ist es wichtig den Laserscanner abzustecken, wenn der Roomba sich an der Ladestation befindet da es sein kann, dass der Laserscanner mehr Akku verbraucht, als die Ladestation fähig ist abzugeben wodurch der Akku des Roombas kaputt geht. Die Notebook, welches auf dem Roomba montiert ist wird über USB Anschluss verbunden. Diese Notebooks sind etwas alt und da auch hier ist die Akkulaufzeit nicht lang ist, dürfen diese nicht lange ohne Ladestation stehen und müssen ebenfalls angeschlossen werden wenn der Roomba in der Ladestation steht.

Unser Roomba ist ein sogenannter Lisener und gibt die Geschwindigkeit linear.x in m/sec und den Drehwinkel angular.z in rad/sec zurück. Das Userprogramm ist sogenannter Publisher für Roomba und Lisener für die Laserdaten. Die Kommunikation findet direkt zwischen Laser bzw. Roomba statt, wird aber durch den roscore ermöglicht.

In Abbildung 1.2 ist der Programmdurchlauf dargestellt. Die while-Schleife, muss in jedem Fall erfüllt werden. Daher ist es besser die Bedingungen die in jedem Fall erfüllt werden müssen in diese Schleife einzuarbeiten. Für die einzelnen Programmteile eignen sich if-Abfrage oder for-Schleifen besser.

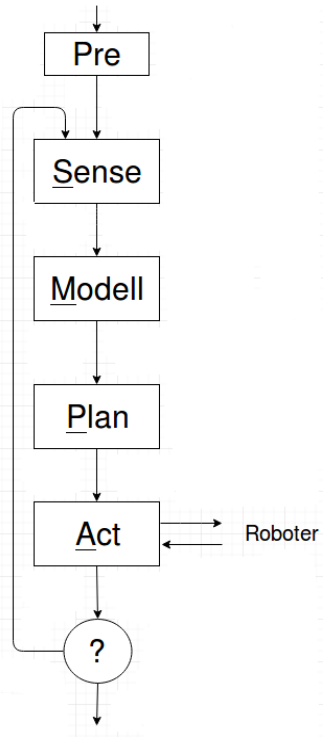


Abb. 1.2

2. Unix

Um mit Unix arbeiten zu können ist es wichtig vorab die grundlegenden Befehle kennen zu lernen um damit umgehen zu können. Vorher aber noch sollte man genau wissen wieso wir eigentlich Unix benutzen und was es ist. Unix ist eine Konsole die uns vor allem das Arbeiten mit dem Robot Operating System, kurz ROS erleichtern soll. Hier einige wichtige Befehle und ihre Bedeutung.

Befehl	**Bedeutung**
$ pwd	"present working directory" : Wo befinde ich mich gerade
$ ls	Auflisten was befindlicher Ordner enthält
$ ls-l	Detaillierte Liste
$ ls-a	Auch versteckte Dateien (.Datei)
$ ls-al	Kombiniert $ls-a und $ls-l
$ ls-s	Kurze Version
$ cd	Change directory
$.	Aktuelle directory
$ ..	Directory vor aktuellem
$ ~	Home directory
$ /	Zeigt an was sich im root Verzeichnis befindet
$ mv	"move" : verschieben
$ cp	"copy": kopieren
$ rm	"remove": löschen
$ mkdir	"make directory": legt Verzeichnis an
$ cat>name	"concatenate": erzeugt eine neue Datei
$ rmdir	"remove directory" löscht Verzeichnis
$ rmdir – r	löscht Verzeichnis und Inhalt
$ history	listet alle Befehle die vorher genutzt wurden nach Nummer auf
$!!	letzten Befehl aufrufen und ausführen
$!342	Befehl Nr. 342 aufrufen und ausrühren
$!l	letzten Befehl der mi l beginnt aufrufen und ausführen

3. ROS

Das Robot Operating System kurz ROS ist ein Open-Source-Betriebssystem, dass von Willow Garage zur Steuerung von Robotern entwickelt wurde. Es dient dazu unterschiedliche Knowhows zusammen zu bringen. Diese Plattform ist für alle frei verfügbar und zugänglich, da das Erstellen von wirklich robusten universeller Roboter-Software eine schwierige Aufgabe darstellt. So können alle weltweit, wie in einem öffentlichen Repository zusammen an einem Problem arbeiten und dieses gemeinsam lösen. Heute besteht ROS aus zahlreichen kleinen Computerprogrammen die miteinander verbunden die Möglichkeit bieten Nachrichten auszutauschen.

In ROS existieren sogenannte Nodes, das sind mehr oder weniger Knoten ausführbarer Dateien innerhalb eines ROS-Pakets. Diese Knoten verwenden eine ROS client library um mit anderen Knoten zu kommunizieren. Je nach Programmiersprache unterscheidet sich die library, da wir in unserer Projektgruppe in C++ arbeiten nutzen wir die roscpp client library. Nodes können zu einem sogenannten Topic publishen ("veröffentlichen") sowie ein solches subscriben ("abonnieren"). Ein

roscore ist der Name eines Service für ROS der Nodes untereinander zu kommunizieren und verschiedene Topics zu subscriben bzw. zu publishen, dieser steht an wichtigster Stelle.
ROS funktioniert am besten unter einer Ubuntu-Linux-Umgebung stellt unter Unix eine Reihe von Befehlen zur Verfügung.

3.1. Grundlagen, Befehle

Um letztendlich über die Konsole mit ROS arbeiten zu können sollten zu Beginn die wichtigsten Befehle verstanden werden. Zunächst erstellen wir uns einen Workspace auf dem wir arbeiten. Das machen wir mit dem Befehl *$mkdir –p ~/catkin_ws/src,* in diesem Fall heißt unser Workspace catkin_ws und der Source Ordner enthält lediglich eine CMakeLists.txt, der Name ds Workspace allerdings ist optional. Nachdem wir den Workspace erstellt haben gehen wir in den Ordner mit *$cd ~/catkin_ws/* und kompilieren das Ganze mit einem *$catkin_make*. An einigen Computern im Praktikum sollte immer beim öffnen einer neuen Konsole im eigenen Workspace ein *$source devel/setup.bash* ausgeführt werden, da sonst nicht klar ist in welchem Ordner man sich befindet. Nachdem wir einen Workspace erstellt haben benötigen wir für unser erstes Programm ein Package. Dies lässt sich durch den Befehl *$catkin_create_pkg <package_name> [depend1] [depend2] [depend3]* realisieren. Dies ist kein gültiger Befehl, da der package_name und die einzelnen depends noch ohne Klammerung einzutragen sind. Dieses Package enthält ebenfalls eine CMakeLists.txt und eine package.xml. In der package.xml sollte noch der Maintainer und die E-Mail hinzugefügt werden. Um auch diese Veränderung abzuschließen, kehrt man zurück in den erstellten Workspace und kompiliert das ganze erneut mit *$catkin_make.* In der package.xml finden sich auch die depends wieder mit denen das Package erstellt wurde und es können dort neue hinzugefügt werden. Auch in der CMakeLists.txt sind diese wieder zu finden. Entscheidet man sich also neue depends einzufügen die man vorher vergessen hatte, so sind diese sowohl in der CMakesLists.txt als auch in der package.xml einzufügen. Nachdem all das geschafft ist, kann man in einem beliebigen Texteditor ein Programm implementieren. Wichtig dabei ist dieses im src-Ordner des Package zu speichern und in der CMakesLists.txt einzufügen. In der launch-Datei sollten alle Nodes aufgerufen werden die gebraucht werden. Hier noch ein Paar Befehle die bei der Fehlersuche hilfreich sein könnten.

Befehl	**Bedeutung**
$rostiopic list	Listet alle aktiven Topics auf
$ristopic echo /topicname	Gibt Nachrichten des Topics aus
$rostopic info	Gibt Informationen zu einem bestimmtes aktives Topic
$roswtf	Listet Warnings und Errors des Prozessen auf
$rosbag record <name.bag> /laserscan	Nimmt Laserscandaten auf speichert in .bag Datei
$rosbag info	Gibt Aufnahme wieder

3.1.1. Rqt

Dieser Befehl wird genutzt um für einen aktiven Prozess anzuzeigen welche Nodes miteinander verknüpft sind. Es ist eine Art Software Framework die von ROS zur Verfügung gestellt wird Die Verknüpfung wird in einer Art Mind-Map dargestellt und angezeigt welche Topics zwischen zwei Nodes gepublished oder auf welche subscribed wird.

3.1.2. map_server

Der map_server ist eine Node zum einlesen einer Karte aus der Disk. Diese Karte wird dann als ROS Service bereit gestellt. In unserem Fall bestand die Karte aus dem LBH_floor1 und wurde für unsere Simulationen in Stage und Rviz genutzt. Standartmäßig kann man diese auch über die Konsole mit dem Befehl *$rosrun map_server map_server mymap.yaml* aufrufen, wobei mymap.yaml den Namen der yaml-Datei der eigenen Map beinhalten soll. In der launch-Datei wird der Map_server wie folgt eingefügt.

```
<launch>
<node pkg="map_server" type="map_server" name="map_server" args="$(find
        AIS_worlds)/LBH_floor_1.yaml">
</node>
</launch>
```

3.1.3. tf

Der tf ermöglicht die Transformation über mehrere Koordinatenrahmen. Wenn zum Beispiel Rviz geöffnet wird, sieht man das die Map nicht dem Ursprung des Koordinatensystems entspricht. Hierbei ist es wichtig die Map am Ursprung zu Platzieren. Zu Beginn sind die Werte der X-Y-Z-Koordinaten und die der Quaterionen auf 0 gesetzt. Damit sich also Koordinatensystem und Map überlappen ließt man die Werte für die Pose des Ursprungs ab. Diese kann man ablesen, indem man die Map im Rviz an den Ursprung des Koordinatensystems (gekennzeichnet durch einen gelben Punkt) schiebt. Die sich daraus ergebenden Werte werden in der Launch Datei als Parameter übergeben. Bsp.:

```
<launch>
<node pkg="tf" type="static_transform_publisher" name="link1_broadcaster"
                                                    args= "33.658 -49.380 0 1.57 0 0 /map /odom 100" >
</launch>
```

3.1.4. Amcl

Der Amcl ist ein Lokalisierungssystem für Roboter, dass Partikelfilter für die Position und Orientierung verwendet. Es benötigt für die Position drei Koordinaten, die X-Y-Z Koordinaten, wobei die Orientierung in Quaternionen angegeben wird. Lokalisiert wird über die Laserdaten, daher benötigt der Amcl eine Map und den Laser. Das Name des Package ist ebenfalls amcl und benötigt eine Reihe von Parametern, unter anderem die Initalpose des Roboters. In unserem Fall konnte sich der Roboter lediglich gerade aus, seiner X-Richtung entlang bewegen und um seine Z-Achse drehen, dies entspricht der Drehung in YAW. Die Bewegung des Roboters in Richtung seiner X-Achse entsprach in unserem Koordinatensystem der Y-Achse und da er sich genauso durch rechts und links Drehung entlang der X-Achse

bewegen konnte waren dies unsere drei Koordinaten die wir als Parameter in der Launch-Datei eingeben mussten. X- und Y-Koordinaten und die Drehung um die Z-Achse. Hier ein Beispiel wie der amcl in die Launch-Datei eingefügt werden sollte. Der Name ist dabei optional, Parameter sind in der im Anhang befindlichen Launch-Datei.

```
<launch>
<node pkg= "amcl" type="amcl" name="amcl" output="screen">
<remap from="scan" to="laserscan"/>
</launch>
```

3.1.5. Rviz

Rviz ist eine visuelle Darstellung der Welt in der sich unser Roomba bewegt. Es ist möglich sich in dieser Simulation Daten von Laser, Amcl-Pointcloud und viele weitere Tools anzuzeigen. Vorher aber sollte die vom map_server zur Verfügung gestellte Map ordnungsgemäß transformiert werden, so dass die Koordinaten mit dem im Rviz vorgegebem Koordinatensystem übereinstimmt. Standartmäßig wird Rviz über den Befehl *$rosrun rviz rviz* ausgeführt. Man kann rviz aber auch automatisch über die Launch-Datei aufrufen und starten. Um Konfigurationen beizubehalten und diese standartmäßig auszuführen, wie das anzeigen der Pointcloud oder der Laserdaten, sollten diese unter einer Datei abgespeichert und in der Launch-Datei unter "args=" aufgerufen werden. Hier ein Beispiel für rviz in der Launch-Datei.

```
<launch>
<node pkg="rviz" type="rviz" name="rviz" args="" >
</launch>
```

4. Erste Programme

Die nachfolgenden Robo-Programme arbeiten alle mit einem Laserscanner.

4.1. Robo_A

Das erste Programm ist robo_A, dieses Programm lässt den Roboter so lange geradeaus fahren, bis er zu nah an einem Hindernis ist und stehen bleibt.

4.2. Robo_B

Robo_B ist bereits etwas komplexer, er fährt geradeaus und weicht nach links aus, wenn ihm ein Hindernis rechts zu nahe kommt. Wenn ihm links ein Hindernis zu nahe kommt bleibt er stehen.

4.3. Robo_C

Robo_C fährt rein zufällig durch die Gegend und hält an, wenn er einem Hindernis zu nahe kommt.

4.4. Robo_D

Robo_D fährt geradeaus und weicht zu beiden Seiten Hindernissen aus, indem er in die entgegengesetzte Richtung dreht. Wenn das Hindernis zu nah ist und er nicht mehr drehen kann bleibt er stehen.

4.5. Wall-finder (robo_E)

Der Roomba hat die Aufgabe zur Wand orientiert mit einem Abstand von einem Meter stehen zu bleiben.

Dies lässt sich beim Roomba mithilfe von drei Laser-Sensoren umsetzen (Laser 250 rechts vorne, Laser 270 geradeaus, Laser 290 links vorne).

Die Bewegungsgeschwindigkeit ist die Differenz aus dem Abstand geradeaus zur Wand (Laser 270) subtrahiert mit dem gewünschten Abstand von einem Meter. Bei einer hohen Differenz fährt der Roomba schneller, bei einer niedrigen langsamer. Sollte der Abstand zur Wand geringer als ein Meter sein, ergibt sich eine negative Differenz. In Folge dessen fährt der Roomba rückwärts. Ist die Distanz zur Wand größer als ein Meter, somit die Differenz positiv, fährt der Roomba vorwärts.

Steht der Roomba nicht frontal zur Wand, muss er sich zu ihr hin orientieren. Die dafür notwendige Drehgeschwindigkeit und Drehrichtung bestimmen wir anhand der Differenz der Entfernung zum Hindernis rechts vorne (Laser 250) subtrahiert mit der zum Hindernis links vorne (Laser 290). Hierbei entspricht eine negative Differenz einer Linksdrehung und eine positive Differenz einer Rechtsdrehung. Je höher die Differenz, desto schneller dreht sich der Roomba.

Die Bewegungs- und Drehgeschwindigkeiten, die benötigt werden, um den Roomba in die gewünschte Pose zu bringen, müssen immer wieder aktualisiert werden. Zur Berechnung dieser Geschwindigkeiten wird in unserem Quellcode die Funktion *calculateCommand()* ausgerufen, die fortwährend in einer Schleife (hier: *mainLoop*) neu berechnet wird.

4.6. Wall-follower (robo_F)

Der Roomba hat die Aufgabe an einer Wand entlang zu fahren.

Hierfür werden zwei Laser benötigt. Ein Laser ermittelt den Abstand zur rechten Seitenwand, zur der der Roomba in einen Abstand von 0,7 m hält. Verändert sich der Abstand, muss der Roomba diesen mittels einer Drehung korrigieren. Diese Drehung berechnet sich aus der Differenz, zwischen dem gewünschten Abstand von 0,7 m und der eigentlichen Entfernung zur rechten Wand. Wird die Differenz größer erhöht sich die Drehgeschwindigkeit. Abhängig vom Vorzeichen der Differenzen kommt es beim Korrigieren zu einer Links- bzw. Rechtskurve.

Der andere Laser misst den Abstand nach vorne. Sollte sich geradeaus vom Roomba ein Hindernis befinden, passt er die Bewegungsgeschwindigkeit der Entfernung zur Wand an. Steuert der Roomba in eine Ecke hinein, wird er immer langsamer, da der Abstand nach vorne immer geringer wird. Wenn der Roomba eine gewisse Geschwindigkeit unterschritten hat, findet eine Linksdrehung statt.

Die Geschwindigkeiten werden in der Funktion *calculateCommand()* berechnet und in der *mainLoop* angepasst.

4.7. Robo_3_0

Robo_3_0 macht das Gleiche, wie Robo_D, benutzt allerdings rviz und amcl um seine aktuelle Position zu ermitteln und anzuzeigen.

4.8. Guess_01

Guess_01 arbeitet statt mit einem Laserscanner mit einer bgr-Kamera. Es bekommt das Kamerabild als Eingabe und berechnet den Mittelpunkt der roten Pixel. Dieser wird auch optisch durch einen Kreis auf dem Kamerabild ausgegeben. Wenn der Mittelpunkt im linken (rechten) Bereich des Bildes ist, dann fährt der Roboter nach links (rechts). Wenn der Mittelpunkt im oberen (unteren) Bereich des Bildes ist, dann fährt der Roboter nach vorne (hinten).

4.9. Person Follower

Der Person Follower besteht aus einem Edge Detector, einem Leg Detector und einem Humanleg Detector.

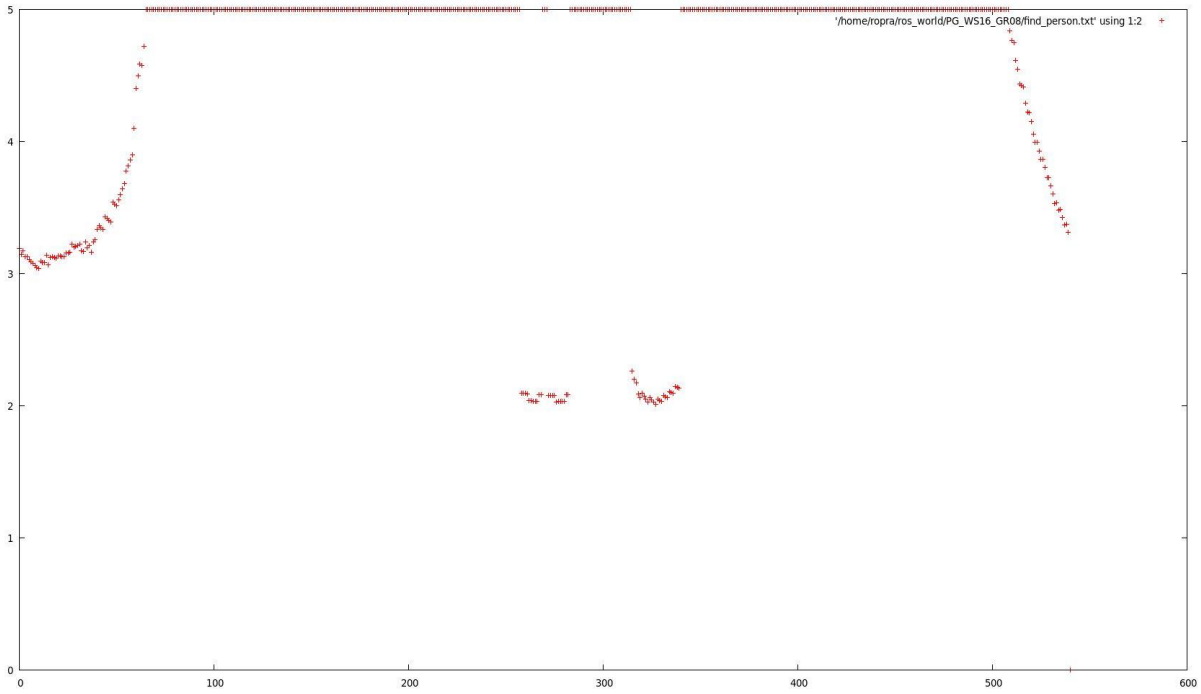


Abb. 4.1

4.9.1. Edge Detector

Der Edge Detector läuft über die Laserdaten (Abb. 4.1) und sucht nach den Kanten, diese werden in einem neuen Array „edges” abgespeichert (Abb. 4.2). -1 für steigende Kante, 1 für fallende Kante und 0 für keine Kante. Dabei werden alle Messungen über 7 m ignoriert um Fehlmessungen abzufangen.

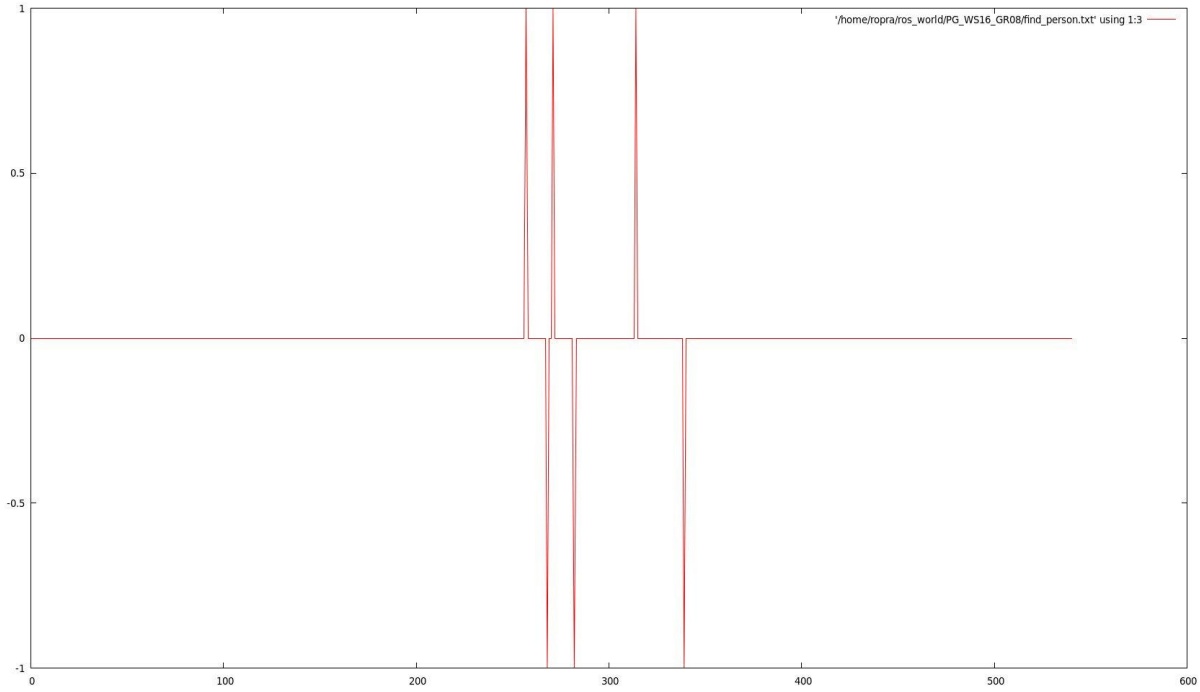


Abb. 4.2

4.9.2. Leg Detector

Der Leg Detector sucht in der Eingabe (Abb. 4.2) nach Beinen, also zuerst nach einer fallenden Kante, gefolgt von einer steigenden Kante. Sein Ergebnis schreibt er in den Array Leg (Abb. 4.3), wobei 1 bedeutet, ist Teil eines Beines und 0 bedeutet ist nicht Teil eines Beines.

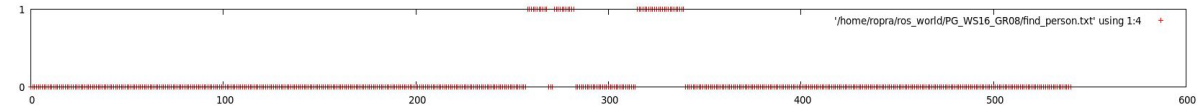


Abb. 4.3

4.9.3. Humanleg Detector

Zuerst wird der Inhalt des Arrays Leg (Abb. 4.3) in den Array Humanleg übertragen. Nun prüft der Humanleg Detector, ob die zuvor gefundenen Beine einen oder mehrere Menschen ergeben können. Dazu werden nur die Beine berücksichtigt, die mindestens 7 cm und maximal 25 cm Durchmesser haben. Alle Beine, die diese Kriterien nicht erfüllen werden aus dem Array gelöscht.

Nun werden 2 Beine nur als Menschenbeine erkannt, wenn sie einen minimalen Abstand von 3 cm und einen maximalen Abstand von 50cm zueinander haben. Auch hier werden wieder alle Beine, die diese Kriterien nicht erfüllen aus dem Array gelöscht (Abb. 4.4).

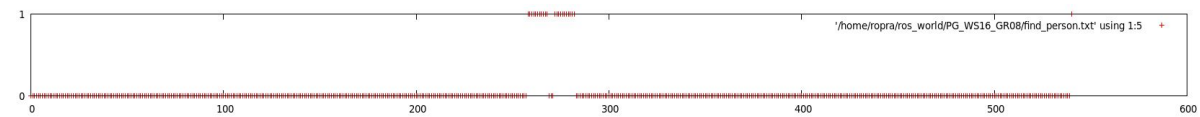


Abb. 4.4

Sind diese 4 Kriterien erfüllt, dann gehören die 2 überprüften Beine zu einem Menschen. Nun wird für den am nächsten stehenden Menschen der Winkel und die Entfernung in angle_of_person und distance_of_person geschrieben.

Der Roboter fährt schließlich mit distance_of_person entlang seiner X-Achse und dreht mit angle_of_person entlang seiner Z-Achse. Dies hat zur Folge, dass, je weiter die Person vom Roboter entfernt ist, bzw. je größer der Winkel zwischen der X-Achse des Roboters und der Person ist, desto schneller fährt/dreht der Roboter.

5. Miniprojekt

Das Ziel des Miniprojekts ist es einen Roboter zu entwickeln, der die Map abfährt, dabei nach Ziffern sucht und sich deren Positionen merkt. Anschließend sollen die Ziffern der Wertigkeit nach abgefahren werden.

5.1. Pose bestimmen

Die aktuelle Pose kann mithilfe der Adaptive Monte Carlo Localization bestimmen werden. Hierzu wurde folgende Funktion geschrieben:

amclCallback(const geometry_msgs::PoseWithCovarianceStamped m_amcl_data)

Das Objekt *m_amcl_data.pose.pose* besteht zum einen aus dem Objekt *geometry_msgs::Point*, das die Position des Roboters beinhaltet und zum anderen aus dem Objekt *geometry_msgs::Quaternion*, das einem die Orientierung des Roboters liefert.

Die Position wird mit *m_amcl_data.pose.pose.position* bestimmt. Für die Abfrage einer bestimmten Koordinate der aktuellen Position, können die Funktionen *m_amcl_data.pose.pose.position.x, m_amcl_data.pose.pose.position.y und m_amcl_data.pose.pose.position.z* verwendet werden.

Die Orientierung des Roboters wird mit *m_amcl_data.pose.pose.orientation* mittels Quaternionen zurückgegeben. Auch in diesen Fall können die vier Komponenten eines Quaternions einzeln zurückgegeben werden. Mit der Funktion *tf::getYaw(m_amcl_data.pose.pose.orientation)* kann die Orientierung, gegeben als Quaternion, ebenfalls als Euler-Drehung um die z-Achse umrechnen werden.

Die Funktion *ros::spinOnce()* soll nun für eine kontinuierliche Aktualisierung des *amclCallback* sorgen. Damit die Position auch stetig aktualisiert wird, ist es wichtig einen Subscriber im Constructor wie folgt zu definieren:

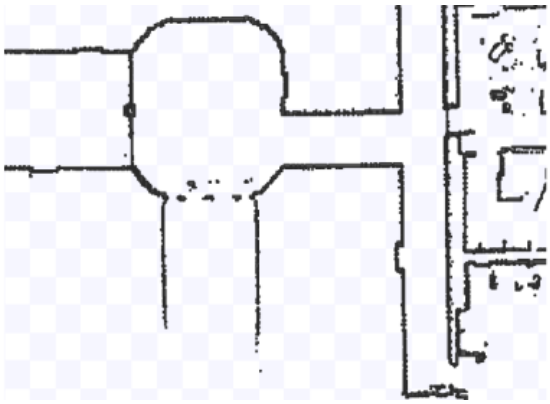
```
Roboter_name::Roboter_name() {

    ros::NodeHandle m_nodeHandle("/");

    m_amclSubscriber =m_nodeHandle.subscribe<geometry_msgs::

        PoseWithCovarianceStamped>("amcl_pose", 20,&Map_mover::amclCallback,this);

}
```

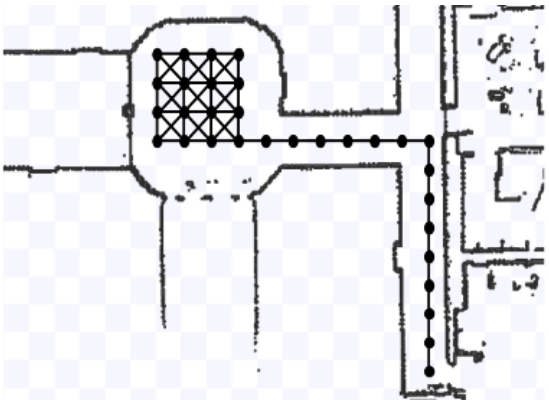
5.2. Lokalisation und Navigation auf einer Karte

Die Kapitel erklärt, wie sich der Roomba auf einer Karte zwischen zwei Punkten frei bewegt.

Für die Lokalisation der aktuellen Position des Roombas verwenden wir den amcl, wie es im Unterpunkt Pose bestimmen gezeigt wurde.

Damit der Roomba unbeschadet navigieren kann, wurde über die Karte ein 1 m Knotennetz gelegt.





Um nun zwischen zwei Knoten zu navigieren, wird mit Hilfe des A*-Algorithmus der kürzeste Weg im Graphen berechnet. Folgt der Roomba diesem berechneten Weg, Knoten für Knoten, wird eine Kollision mit Hindernissen auf der Karte verhindert. Hierbei werden benachbarte Knoten im Graphen auf direktem Weg angefahren. Dies geschieht mit folgender void Funktion:

Map_mover::moveGoal(geometry_msgs::Pose goal_pose, double distance_tolerance)

Damit der Graph mit dem 1 m Raster erstellt werden kann, wurde folgende Funktion geschrieben:

Graph::build_graph()

Diese Funktion bezieht sich auf zwei Textdateien (map1.txt und map2.txt) und gibt einen fertigen Graphen zurück. Die Textdatei *map1.txt* enthält in der ersten Zeile die Knotenanzahl und in den folgenden Zeilen jeweils die Knoten-ID gefolgt von den x und y Koordinaten auf der Karte. Die Textdatei *map2.txt* enthält pro Zeile eine Kante, welche durch die zwei entsprechenden Knoten-IDs dargestellt wird. Diese Kanten sind ungerichtet zu betrachten.

Damit der Roomba nun zwischen zwei Knoten navigiert, setzt man den A*-Algorithmus ein. Hierzu wird als zulässige Heuristik der euklidische Abstand verwendet. Aus der Zulässigkeit der Heuristik, ist die Optimalität des A*-Algorithmus gegeben. Mit der Funktion: *Graph::a_star(Graph::NodeId start, Graph::NodeId goal)* wird der kürzeste Weg vom start-Knoten zum goal-Knoten gesucht und mit einem *vector<Graph::NodeId>* zurückgegeben.

Pseudo-Code des A*-Algorithmus:

Eingabe: Graph G, Startknoten s, Zielknoten d

Ausgabe: Weg vom Startknoten zum Zielknoten oder Fehler

(1) Open = {s}, Closed = {}
(2) If Open ist leer then return Fehler
(3) Nehme Knoten n aus Open mit minimalen f(n)
(4) Füge n Closed hinzu
(5) If n ist Zielknoten then return Weg vom Startknoten zum Zielknoten (mithilfe von Pointern)
(6) Für alle Nachfolger n‘ von n in G
    1. If n‘ weder in Open noch in Closed then n‘ speichert Pointer auf n und füge Open hinzu
    2. If n‘ in Open oder Closed und der Weg über n kürzer als vorher then n‘ speichert Pointer auf n

(7) Gehe zu Schritt (2)

Unsere C++-executable move.cpp muss in der CmakeLists.txt noch deklariert werden. Damit wir unsere geschriebene Graphenklasse in der executable verwenden können, müssen wir den entsprechenden .cpp File der Graphenklasse wie folget übergeben:

```
add_executable(move src/move.cpp src/graph.cpp)
target_link_libraries (move ${catkin_LIBRARIES} )
```

Um Funktionen aus graph.cpp in move.cpp aufrufen zu können, fehlt ein include der Header-Datei in move.cpp (#include "graph.h").

5.3. Orientierung zum Scan

Der Roomba hat die Aufgabe die Karte entlang zu fahren und Zahlen zu lesen, um diese anschließend in richtiger Reihenfolge abzufahren. Im Kapitel Lokalisation und Navigation auf einer Karte wurde gezeigt, dass sich der Roomba auf der Karte entlang der Knoten frei bewegen kann. Um nun die Wände nach Zahlen abzusuchen, fährt der Roomba jeden Knoten im Graphen ab. Damit nicht zu viele Zahlen doppelt gescannt werden, haben wir einen Algorithmus geschrieben.

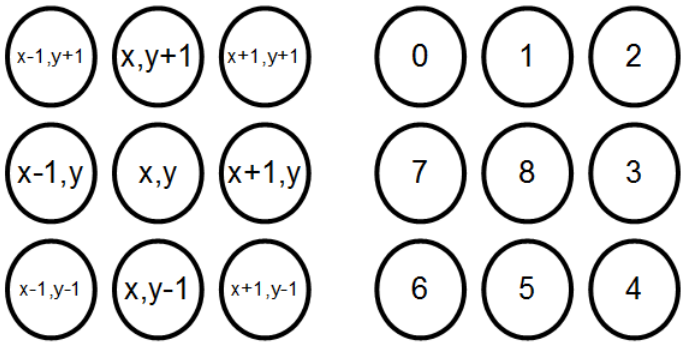
Pseudo-Code des Algorithmus Orientierung zum Scan:

Eingabe: Graph G, Position vom Roomba

Ausgabe: Zu scannende Knotenmenge

- S={0,1,2,3,4,5,6,7}; N={}; D={}
- For i := 0 to 7 do:
  - If i ist Nachbarknoten von 8
    - N = N U {i}
  - If i mod 2 == 1 und i ∈ N
- D = D U {i-1}; D = D U {(i+1) mod 8}
- If i mod 2 == 0 und i ∈ N
- D = D U {i+1}
- S = S\(D U N)
- Return S

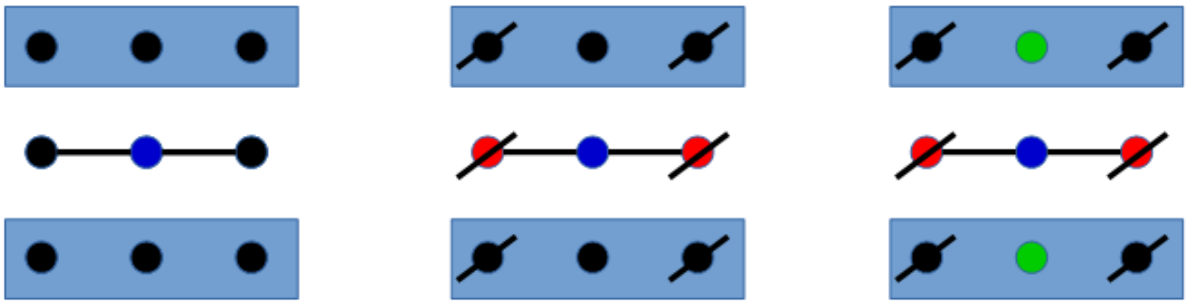
Die nachfolgende Graphik veranschaulicht die Positionen der Knoten auf der Karte.



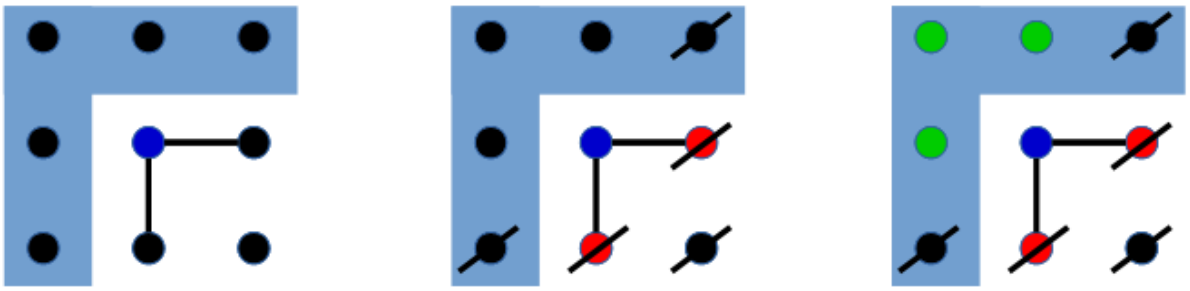
Auf der Graphik entspricht Knoten 8 (x, y) der Position des Roombas innerhalb des Graphen. Die anderen Knoten entsprechen den verschiedenen Positionen im Graphennetz.

Die Idee des Algorithmus ist es, wenn ein Nachbarknoten von Typ 1, 3, 5 oder 7 im Graphen existiert, so wird dieser Knoten von Typ x und die Knoten von Typ x-1 und (x+1) mod 8 gestrichen. Zurück bleiben die Knoten, an denen die zu scannenden Wände sind. Im Folgenden sind die Beispiele eines Tunnels, einer Ecke und einer Sackgasse aufgeführt.

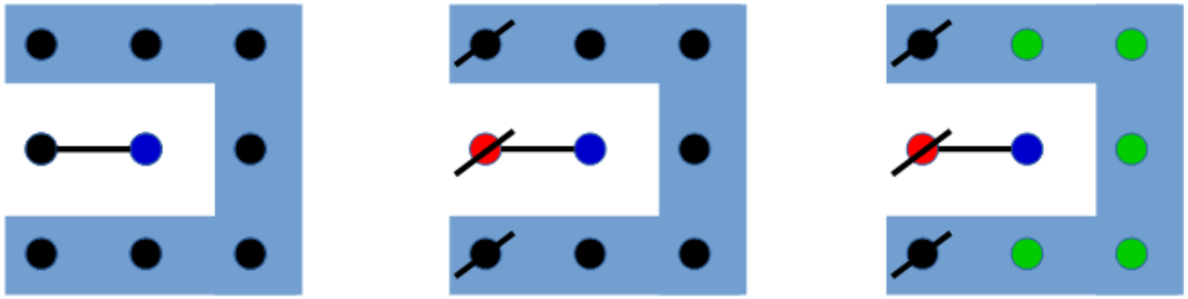
Tunnel:



Ecke:

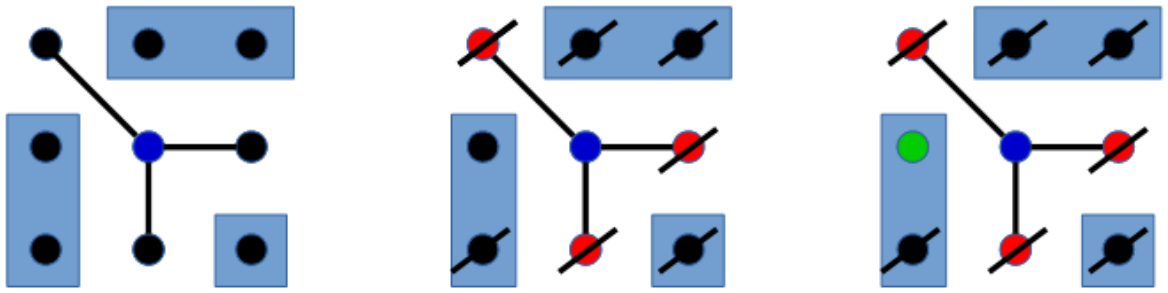


Sackgasse:

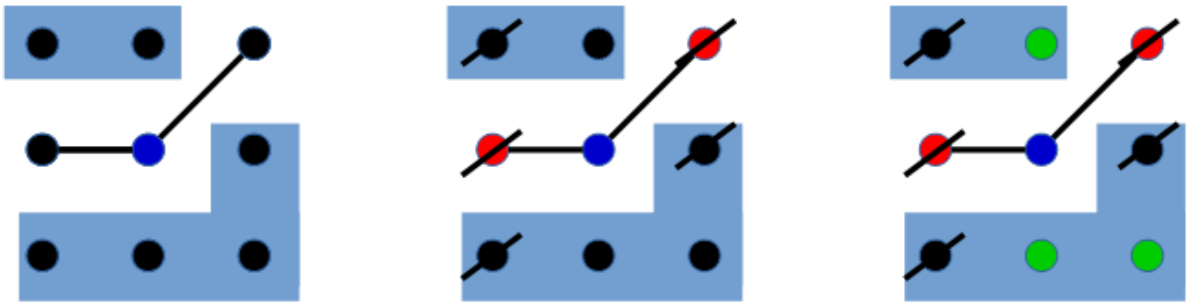


Sollte das Graphennetz später erweitert werden, kann es zu Sonderfällen kommen, bei denen nur eine diagonale Kante existiert. Damit in diesem Fall nicht doppelt die Wand nach einer Zahl abgescannt wird, betrachtet der Algorithmus nur die Wand mit dem Knoten links von der Kante (aus Sicht des Roomba). Der Knoten rechts von der Kante wird gestrichen. Dieser wird nur gescannt, wenn sich der Rommba am Nachbarknoten der Kante befindet.

Sonderfall 1:



Sonderfall 2:

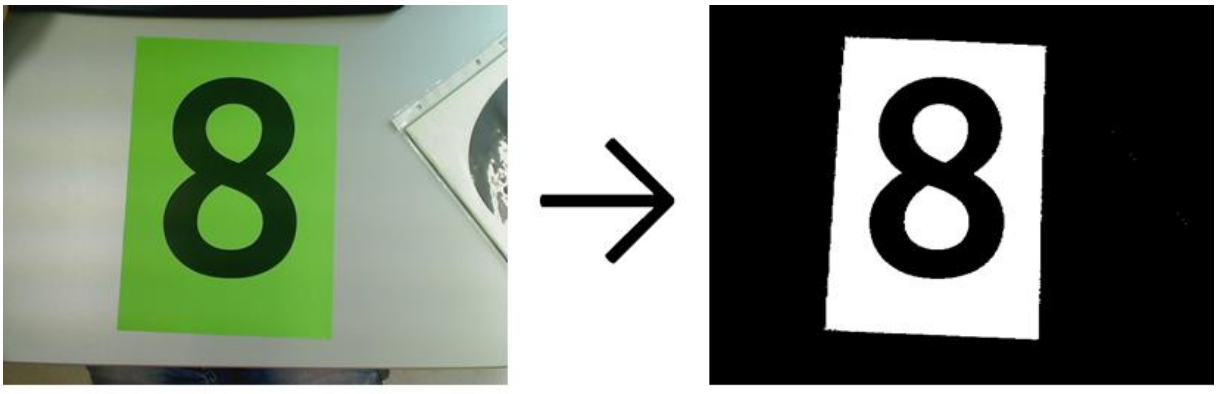


5.4. Filter

Bevor Ziffern gefunden werden können wird das Kamerabild gefiltert. Dies geschieht mit den 3 folgenden Filtern.

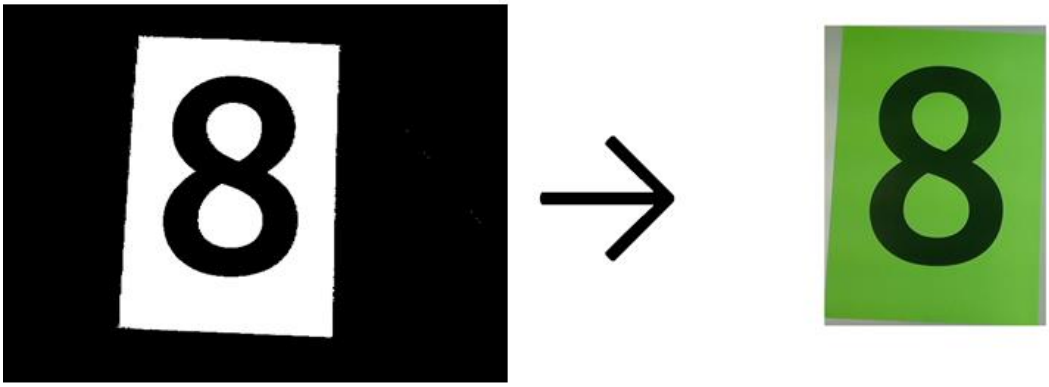
5.4.1. Detect_green_areas

Der detect_green_areas Filter bekommt ein Bild als Eingabe und setzt für alle Pixel mit bestimmten gbr-Werten (bestimmtes grün) die entsprechenden Pixel in einem Ausgabebild auf weiß (255) und alle anderen Pixel auf schwarz (0).



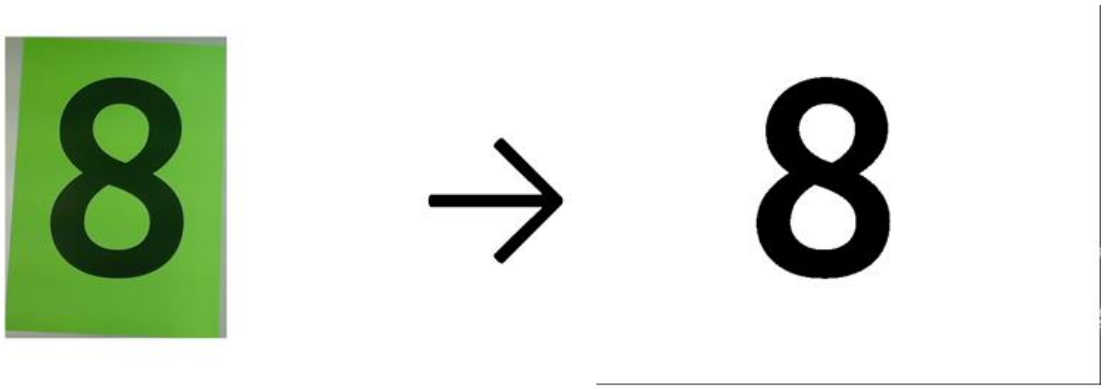
5.4.2. Find_quadrilaterals

Der find_quadrilaterals Filter erzeugt ein neues weißes Bild in Größe des Kamerabildes und sucht in dem Eingabebild nach Vierecken. Für jedes Viereck ermittelt er den minimalen und maximalen X und Y Wert und fügt in dem weißen Bild in dieser Fläche das Originalbild ein.



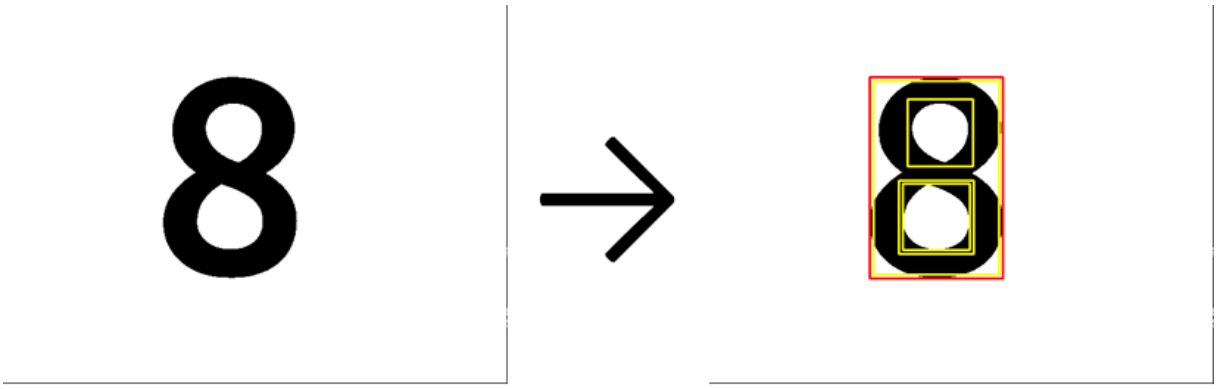
5.4.3. Make_white_and_black

Der make_white_and_black Filter erzeugt aus dem Eingabebild ein Graustufen-Ausgabebild, wobei jedes Pixel, dessen Grünwert größer 100 ist auf weiß und jedes andere Pixel auf schwarz gesetzt wird.



5.5. OCR

Die OCR speichert sich zuerst ein Trainingsmuster bestehend aus den 10 verschiedenen Ziffern. Sie sucht anschließend im Eingabebild nach Konturen aus mindestens 5000 Pixeln (gelb umrandet). Auf die größte gefundene Kontur (rot umrandet) wird der k-Nearest-Neighbour-Algorithmus angewendet, es wird also die gefundene Kontur mit dem Trainingsmuster abgeglichen. Anschließend wird die größte Übereinstimmung als Ergebnis zurückgegeben.



In diesem Beispiel wurde eine 8 als Ergebnis zurückgegeben.

5.6. Wie gut funktioniert die Ziffernerkennung?

Wenn die Lichteinstrahlung zu stark oder zu schwach ist wird der Grünton so stark verfälscht, dass er vom detect_green_areas Filter nicht mehr erkannt wird und das Ausgabebild komplett schwarz ist, wodurch am Ende keine Ziffer gefunden wird. Des Weiteren werden nur Ziffern größer 5000 Pixel erkannt und ungültige Zeichen werden nicht raussortiert, sondern als die nächste Ziffer erkannt.

Eine Messung vom 20.04.2017 20:30 Uhr ergab folgendes Ergebnis.

Zahl	Richtig erkannt	Fälschlicherweise „erkannt“
0	56.5%	0%
1	96.4%	6.9%
2	87.1%	0.0%
3	100.0%	14.8%
4	87.0%	9.1%
5	80.8%	4.5%
6	57.9%	31.3%
7	100.0%	7.7%
8	95.2%	45.9%
9	96.0%	0.0%

6. Quellen

http://wiki.ros.org/amcl [20.04.2017]

http://wiki.ros.org/map_server [20.04.2017]

http://wiki.ros.org/tf#static_transform_publisher [20.04.2017]

http://wiki.ros.org/catkin/Tutorials/create_a_workspace [26.04.2017]

http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes [26.04.2017]

http://wiki.ros.org/rviz/UserGuide [27.04.2017]

Buch: Programming Robots with ROS- A Practical Introduction to the Robot Operating System, Autoren: Morgan Quigley, Brian Gerkey, William D. Smart, Verlag: O'Reilly Media, Ausgabe: November 2015