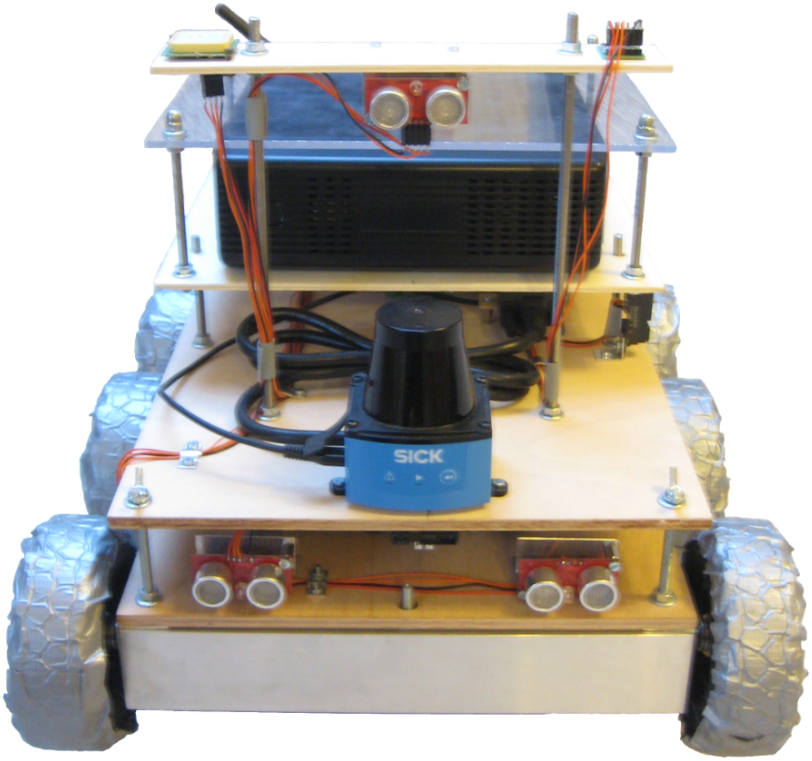
Besondere Lernleistung

Fach Informatik/Physik/Mathematik

Bau und Programmierung eines autonomen Erkundungsroboters



Mark Springer

Jgst. 13

Betreuender Lehrer: Herr Fassbender

Inhaltsverzeichnis

Abbildungsverzeichnis

Vorwort

Das Thema dieser besonderen Lernleistung, im Rahmen des Abiturs, ist die Konstruktion und Programmierung eines autonomen Roboters, welcher nach Angabe einer GPS-Koordinate, selbstständig zu dieser navigiert, während er allen Hindernissen ausweicht. Es gibt mehrere Gründe, warum wir uns dafür entschieden haben. Zum einen ist es ein angemessen anspruchsvolles Thema, zum anderen findet die gebaute Roboterplattform auch über den Rahmen der Lernleistung hinaus Verwendung. Daher wollen wir sie auch in Zukunft erweitern und entwickeln. Der Grund, warum mein Bruder Jannik und ich uns entschieden haben eine Besondere Lernleistung zu absolvieren, ist neben der Abiturnote ein ausgeprägtes Interesse an der Elektronik und der Programmierung. An dieser Stelle möchte ich unseren Sponsoren danken. Als erstes ist die Stiftung Evolution zu nennen, die uns großzügig bei der Anschaffung, der nicht immer billigen Teile der Roboterplattform unterstützt hat. Wir danken auch der Firma Sick, die uns freundlicherweise einen ihrer Laserscanner zur Verfügung gestellt hat.

Und natürlich sind wir dankbar für die Hilfe und Unterstützung unserer Eltern.

Der Text wurde mit LaTeX erstellt.

1. Abgrenzung des Themas

Um den Gegenstand der Arbeit weiter abzugrenzen, möchte ich hier kurz skizzieren, was nicht zu der Zielsetzung von meinem Bruder und mir gehört hat. Zuerst ist da die Erstellung einer elektronischen Karte, anhand derer navigiert werden kann, zu nennen, die wir letztendlich nicht umgesetzt haben. Dafür gibt es mehrere Gründe.Zum einen hätte ed den Rahmen der Besonderen Lernleistung gesprengt.

Desweiteren gibt es bis heute keinen Algorithmus, der die gleichzeitige Erstellung und Navigation mit einer Karte perfekt realisiert. Das liegt daran, dass das Programm auf dem Roboter um eine Karte zu erstellen seine Position kennen müsste. Um jedoch die Position ohne benutzung von GPS zu berechnen wird eine Karte benötigt. Dies ist auch als das Problem der gleichzeitigen Lokalisierung und Kartierung bekannt oder SLAM (simultaneous localization and mapping). Natürlich kann die Position auch mittels GPS bestimmt werden, aufgrund von variierenden Positionswerten und Genauigkeit, der GPS-Koordinaten, kommt dies allerdings nicht infrage, da wir auf der Ebene von unter einem Meter arbeiten, dazu später mehr. Eine andere Möglichkeit ist die allmähliche Erstellung der Karte anhand von markanten Merkmalen des durchfahrenen Raums, welche eine Orientierung ermöglichen. Dies ist jedoch, wie erwähnt, ein enormer Aufwand, welcher nicht in den Rahmen einer Lernleistung passt. Aus diesem Grund habe ich mich für einen rein reaktiven Algorithmus für das Umsteuern von Hindernissen entschieden, und nicht für die Erstellung einer Karte. Dies bedeutet, dass der Algorithmus direkt aus den gegebenen Sensordaten eine Ausweich-Richtung errechnet, außerdem wird die GPS-Navigation immer die Luftlinie zum Ziel vorgeben, nicht etwa eine geplante Route. Dies wiederum hat zufolge, dass der Roboter nicht mit Labyrinthen oder labyrinthähnlichen Situationen zurechtkommt, und wenn, dann nur durch Zufall.

1.1. Stand des Problems

Wie bereits angedeutet, ist die autonome Navigation von Robotern ein aktueller Gegenstand der Forschung, in dem Gebiet der Robotik. Die autonome Navigation bei vielen verschiedenen Anwendungen gewinnt zunehmend an Bedeutung, wie zum Beispiel bei Militärrobotern oder Katastrophenrobotern, die alleine durch schwie-

riges Terrain finden müssen, um ihren Zweck zu erfüllen. Offensichtlich lässt sich das Problem in zwei Bereiche aufteilen. Zum einen gibt es die Mechatronik-Seite, sie beinhaltet die Sensorik, Elektronik und die eigentliche Roboterplattform und bedingt somit wie gut der Roboter in der Lage sein wird Hindernisse zu überwinden. Der andere Bereich ist thematisch der Informatik untergeordnet und besteht aus der Programmierung der Steuerungs-Algorithmen und somit auch der Datenverarbeitung. Ich behandele hier letzteren, wie an folgender Grafik ersichtlich wird.

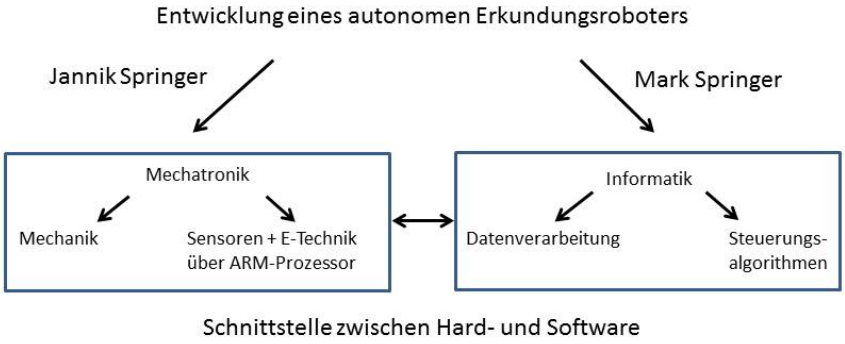


Abbildung 1.1.: Aufgabenverteilung

2. Sensorik

Bevor ich auf die Algorithmen eingehe, möchte ich die verwendete Sensorik, ihre Funktionsweise und die Softwarerepräsentation der Daten die sie liefern, beschreiben.

2.1. Sick Laserscanner

Als wichtigsten Sensor gehe ich zunächst auf dem Sick Laserscanner ein.

In dem Gehäuse rotiert ein Spiegel, der die Laserstrahlen ablenkt, dabei werden von einem Winkelkodierer in regelmäßigen Winkelabständen die Messungen ausgelöst. Dieser Winkelabstand beträgt beim Typ Tim310 werksmäßig 1°, wobei ein Winkelsektor von 270° erfasst und mit 15Hz abgetastet wird. Die maximale Messreichweite beträgt 4 m. Bei einer Auflösung von 1° besteht ein Messpunkt aus einer Mittlung von 84 Einzelmessungen wie im folgenden Bild 2.2 dargestellt:

Abbildung 2.1.: Tim 310 Laserscanner

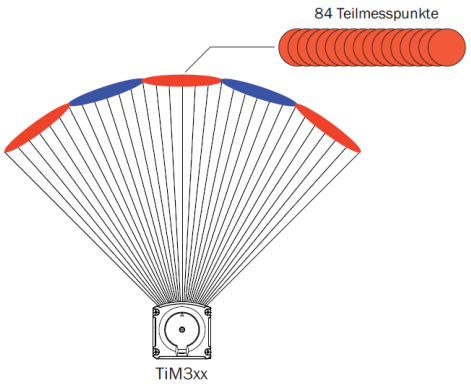


Abbildung 2.2.: Messpunktzusammensetzung

Das eigentliche Messverfahren, genannt HDDM-Technologie (High Definition Distance Measurement) ist patentiert, weshalb es keine weiteren Details dazu gibt. Allerdings handelt es sich um ein Laufzeitmessverfahren, woraus wir schließen können, dass sich der Abstand s aus der gemessenen Laufzeit Δt wie folgt ergibt: $s = \frac{\Delta t}{2} \cdot \frac{c}{n}$ Wobei n der Brechungsindex von Luft ist, der die Lichtge-

schwindigkeit marginal verlangsamt und der Faktor $\frac{1}{2}$ muss eingefügt werden, da das Licht die Strecke Laserscanner-Hindernis zweimal durchläuft.

Wie der Laserscanner Messpunkte behandelt bei denen z.B. die Hälfte der 84 Einzelmessungen einen stark von den anderen Werten abweichenden Wert hat ist mir nicht bekannt und ist Teil der HDDM-Technologie.

Die digitale Repräsentation besteht aus 271 unsigned 16-Bit Integer, wobei, von links beginnend, jeweils ein Messpunkt für einen 1° Schritt steht. Der 136. Messpunkt ist demnach die Entfernung zu einem eventuellem Hindernis direkt vor dem Laserscanner. Die Distanz wird in Millimetern angegeben. Auf welche Weise man an diesen 271 Messpunkt eine Gradskala anlegt, sollte man, aufgrund der vielen Möglichkeiten, vom Verwendungszweck abhängig machen. In meinem Fall interpretiere ich die Messpunkte als, von links beginnend, zu den Graden von -135° bis +135° korrespondierend.

2.2. GPS

2.2.1. Technik

An dieser Stelle werden kurz die Prinzipien der Ortsbestimmung mittels GPS dargelegt.

In Theorie ist es möglich, mithilfe von Drei Messungen zu Drei Satelliten, eine Positionsbestimmung vorzunehmen. Bei einer einzelnen Messung kann sich der GPS-Empfänger auf einer Kugeloberfläche mit dem Mittelpunkt der Satellitenposition und dem Radius der Entfernung Satellit-Empfänger befinden. Die Satellitenposition wird im GPS-Signal mitgesendet.

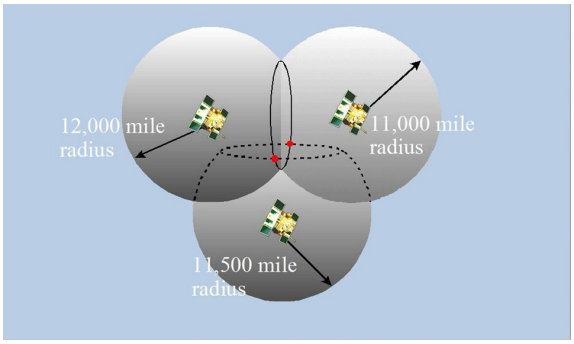


Abbildung 2.3.: 3 Satelliten [1]

Bei der zweiten Messung kommt eine zweite Kugeloberfläche dazu. Der Empfänger muss sich auch auf dieser befinden, das bedeutet, es kommt nur noch die kreisförmige Schnittlinie der beiden Kugeln infrage. Die dritte Messung beschränkt die möglichen Positionen auf zwei Schnittpunkte der drei Kugeloberflächen, von denen einer als unrealistisch ausgeschlossen werden kann, da er sich entweder im inneren der Erde oder irgendwo in der Ionosphäre befindet. In der Praxis gibt es an dieser Stelle ein Problem, da die Entfernung und damit der Radius nicht direkt bestimmt werden kann.

Normalerweise könnte man, ausgehend von der Lichtgeschwindigkeit, die Distanz zu $s = c * (t_{sende} - t_{empfang})$ bestimmen, mit c als Lichtgeschwindigkeit und t_{sende} als die vom Satelliten mitgelieferte Sendezeit in GPS-Zeit.

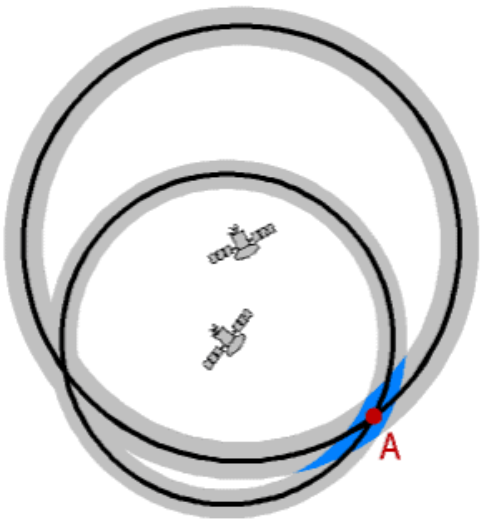
Da es sich um ein unidirektionales Ortungssystem handelt, dass heißt, der Empfänger ist passiv und nur die Satelliten senden, sind die Uhren in Satellit und Empfänger nicht synchronisiert. Daher kann die Zeitdifferenz nur indirekt gemessen werden. Aus diesem Grund sind vier Messungen nötig, um vier Gleichungen aufstellen zu können, mit der Sendezeit des Signals in Satellitenzeit als vierter Unbekannten neben den drei Orts-Unbekannten. Nun sind Sende- und Empfangszeitpunkt in den Gleichungen zwar immer noch in verschiedenen Zeitsystemen, der Unterschied fällt jedoch durch Differenzenbildung bei der Lösung des Gleichungssystems weg.

2.2.2. Störungen

Es gibt mehrere Faktoren, die beeinflussen, wie genau eine gegenwärtig gemessene GPS-Position ist. Zum einen wird das Signal in der Ionosphäre verlangsamt. Je nach Sonnenaktivität sind die Luftschichten mehr oder weniger stark ionisiert, was die Verzögerung hervorruft. Dieser Effekt wird heutzutage weitestgehend durch Berechnungen des Empfängers ausgeglichen, indem der Laufzeitunterschied zweier Signale mit verschiedenen Frequenzen gemessen wird, der in dispersiven Medien, wie die ionisierte Luft in der Ionosphäre, antiproportional zum Quadrat der Frequenz ist. Mit dieser Information kann die Verzögerung ausgeglichen werden.

Eine weitere mögliche Störung ist die sogenannte Mehrwegausbreitung (Multipath Effect). Dieser Effekt kommt durch die Reflexion an nahen Gebäuden, Bäumen oder Ähnlichem zustande. Der Fehler kann im schlimmsten Fall einige Meter betragen und um ihn zu vermeiden bleibt meist nichts anderes übrig als offeneres Gelände aufzusuchen.

Die dritte wichtige Störungsquelle ist eine dem GPS-System immanente, denn sie hängt mit der Satellitenpostion zusammen.



Sind die Satelliten gleichmäßig über dem Himmel verteilt, so ist die Ortung am besten vorzunehmen, sind jedoch alle Satelliten im gleichen Himmelssektor, erschwert dies die Ortung oder macht sie gar unmöglich. Denn wenn man sich zwei Satelliten am selben Ort vorstellt (theoretischer Fall), so sind auch die beiden Kugeloberflächen auf denen sich der Empfänger be-

finden kann identisch und es kann keine Information daraus gewonnen werden. In gleichem Maße wird die Ortung erschwert, wenn die Satelliten nur ungefähr die gleiche Position über dem Himmel haben, da so die beiden Kugeln, wie in Abbildung 2.4 blau angedeutet, einen flacheren Schnittwinkel und damit einen größeren Schnittbereich haben, als wenn sie sich weiter auseinander befänden und einen optimalen Schnittwinkel von 90° hätten.

3. Freiraumnavigation

Die Freiraumnavigation ist ein reaktiver Algorithmus, der aus einem gegebenen Scan-Set direkt einen Wert proportional zur zu fahrenden Richtung berechnet wird. Dabei soll er die bisherige Richtung möglichst beibehalten, während er in freien Raum hinein steuert. Für diesen Zweck wird eine einfache Fuzzy-Regelung verwendet, die im folgenden näher erläutert wird. Da im Bereich der Fuzzy-Logik eine einheitliche Terminologie nicht vorhanden ist, werde ich mich an die Terminologie aus dem Buch "Mobile Roboter"[3] und Wikipedia[4] halten.

Ein Fuzzy-Regler ist zunächst definiert durch vier Elemente:

1. Grundmenge(n) → scharfe Eingangssignale
2. Fuzzy-Prädikate (unscharfe Eingangssignale) → definiert durch Zugehörigkeitsfunktion $\mu(x)$
3. Regelbasis → Regeln WENN...DANN; WENN...UND...DANN (→ unscharfe Stellgrößen)
4. Verfahren zur Defuzzyfizierung (→ scharfe Stellgrößen)

Die scharfen Eingangsvariablen, die Eingangssignale des Reglers, sind in diesem Fall die Werte des Laserscanners und die entsprechenden Richtungen der einzelnen Messwerte. Zu Beiden muss nun eine Grundmenge festgelegt werden, welche jeweils durch den physikalischen Messbereich gegeben ist, demnach von 0 bis 4 Meter und von -90 bis +90°.

Es geht deshalb nur von -90 bis +90°, weil für die Hindernisvermeidung die Objekte hinter dem Roboter keine Rolle spielen, das heißt es werden nur 180 der 270 Messwerte des Laserscanners benutzt.In Grundmengen können nun unscharfe Teilmengen selbiger definiert werden. Im Gegensatz zur klassischen Mengenlehre ist es in der Theorie der Fuzzy-Mengen möglich, dass ein Element nur mit einer gewissen Signifikanz, auch Zugehörigkeitsgrad, einer unscharfen Teilmenge angehört. Sie werden durch eine Zugehörigkeitsfunktion definiert, die jedem scharfen Eingangssignal einen Zugehörigkeitsgrad zuordnet. Dabei muss für $\mu(x)$ in dem Intervall der Grundmenge gelten: $0 < \mu(x) < 1$. Aufgrund des erklärten Ziels in den freien Raum zu steuern und dabei möglichst die bisherigen Richtung beizubehalten, ergeben sich zwei Fuzzy-Prädikate mit der linguistischen Bezeichnung “ist frei“ und “ist

in Fahrtrichtung“. Die Definition von Prädikaten ist ein Schritt, bei dem Entwurf eines Fuzzy-Reglers bei dem Expertenwissen einfließen muss, um durch geeignetes Modellieren einer Zugehörigkeitsfunktion den Fuzzy-Prädikaten die Bedeutung und Funktion zu geben, die ihnen durch ihre linguistische Bezeichnung zugedacht wurde. Ich werde dies nun beispielhaft anhand der beiden gegebenen Fuzzy-Prädikate erläutern, dazu sei das Prädikat “ist in Fahrtrichtung“ die Variable A und das Prädikat “ist frei“ die Variable B.

A ist, wie in Abbildung 3.1 dargestellt, mit der Funktion $\mu(\phi_i)_A = -k * \phi_i^2 + 1$ realisiert, sodass die Richtungen nahe Null hohe Werte erhalten und Richtungen am Rand der Grundmenge niedrigere. Die Variable ϕ wird, wie auch hier, häufig für Winkelparameter benutzt.

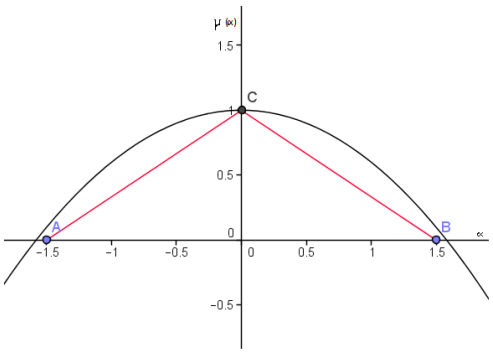


Abbildung 3.1.: Prädikat A

Auf diese Weise haben Richtungen, die wenig oder nicht von der Aktuellen abweichen einen höheren Zugehörigkeitsgrad, das heißt, sie gehören sehr stark zu der unscharfen Teilmenge “ist in Fahrtrichtung“. Die Grundmenge ist hier im Bogenmaß angegeben, also etwa von -1.5 bis +1.5 für -90 bis +90°.

Eine andere Möglichkeit wäre eine lineare Definition über ein Dreieck (in Bild 3.1 rot dargestellt), da jedoch der Aufwand identisch ist, habe ich mich für die elegantere quadratische Funktion entschieden.

Der Parameter k hat aktuell den Wert 0.4, was bedeutet, dass $\mu(x)$ bei -90 bis +90° nicht ganz Null ist, dies spielt jedoch bei der Navigation keine Rolle.

Das Prädikat B hat die Zugehörigkeitsfunktion $\mu(x)_B = \frac{1}{\pi} * atan(k * (x - d)) + 0.5$. Der Arkustangens wird in der Zugehörigkeitsfunktion verwendet, da er einen gleichmäßigeren Übergang von nah zu fern bietet, als die lineare Möglichkeit (in Bild 3.1 rot).

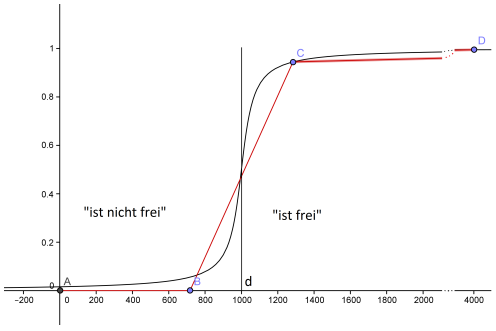


Abbildung 3.2.: Prädikat B

Wie man sehen kann werden alle Entfernungen größer d als “ist frei bewertet“, das heißt mit einem Zugehörigkeitsgrad größer als 0.5, und alle kleiner d nur mit einem Zugehörigkeitsgrad von kleiner als 0.5. Bei der aktuellen Implementation hat sich d = 1000 mm als sinnvoll erwiesen. K lässt den Übergang entweder steiler oder sanfter werden und hat in der Implementation den durch Erfahrung gewonnenen Wert 0.02.

Als nächstes gilt es Regeln zu definieren, welche die unscharfen Eingangsvariablen in eine unscharfe Stellgröße (Ausgangsvariable) umwandeln. Wie bereits erwähnt, haben die Regeln die generelle Form einer Implikation, das heißt, sie bestehen aus einer oder mehreren Prämissen, dem WENN Teil und aus einer Konklusion, dem DANN Teil. Da wir zwei Eingangsvariablen haben, müssen diese durch einen Operator verknüpft werden, in diesem Fall dem UND Operator, weil der Roboter in eine Richtung fahren soll, wenn sie “frei“ UND “in Fahrtrichtung“ ist.

Diese Regeln haben offensichtlich die Form:

WENN “in Fahrtrichtung“ UND “ ist frei“ DANN “fahre in diese Richtung“.

Diese Regel muss auf jedes Wertepaar (Winkel,Entfernung) angewendet werden. Das Fuzzy-UND ist über eine einfache Multiplikation implementiert, um der Funktion $\mu_A(x)$ einen gerichteten Wert zu geben. Da das Vorzeichen durch die Quadierung verloren geht, muss zusätzlich noch mit $sin(\phi)$ multipliziert werden. Das Produkt $X(\phi, x) = sin(\phi) * \mu_A(\phi) * \mu_B(x)$ ergibt den Graphen 3.3, unter der Bedingung, dass die Umgebung des Roboters in allen Richtungen frei ist, das heißt $\mu_B(x)$ immer 1 ist.

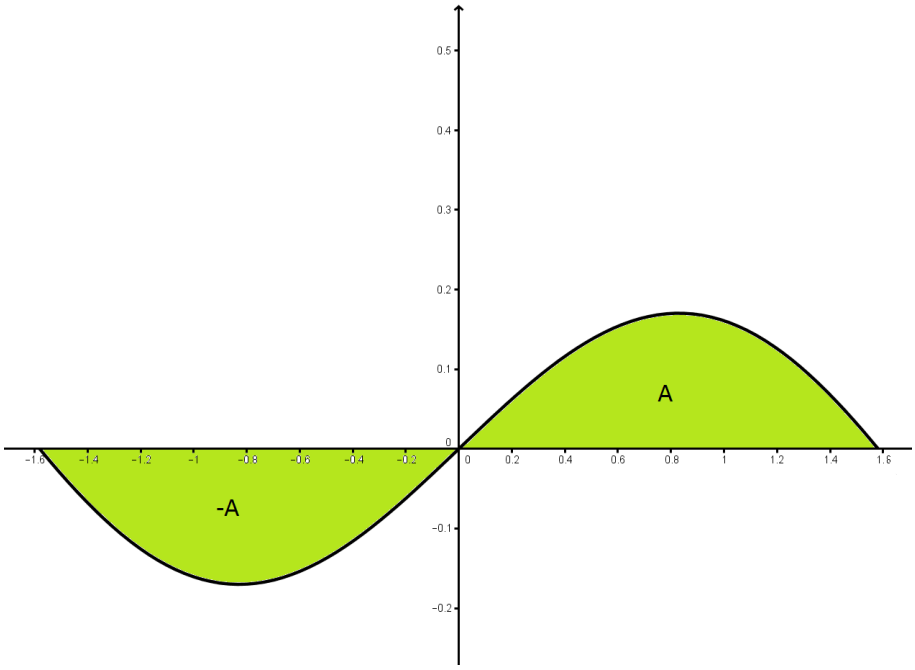


Abbildung 3.3.: $X(\phi, x) = sin(\phi) * \mu_A(\phi) * \mu_B(x)$

Um nun einen Wert für die Ausweichrichtung zu bekommen, werden die 180 Wertepaare des Laserscanners in dieses Produkt eingesetzt und aufsummiert, um so einen Wert für die Fläche zu erhalten. Da beide Flächen unter dem Graphen gleich groß sind, jedoch mit unterschiedlichem Vorzeichen, ergibt sich die Summe 0. Das heißt die aktuelle Richtung kann beibehalten werden. Taucht nun, wie in Bild 3.4 dargestellt auf der rechten Seite des Roboters ein Hindernis auf, so ergibt sich, wenn einige

Werte von $\mu_B(x)$ kleiner 1 sind, eine Lücke im Graphen von p(x). Dadurch wird nun, wenn man die Summe bildet, das Ergebnis negativ und die aktuelle Richtung muss geändert werden.

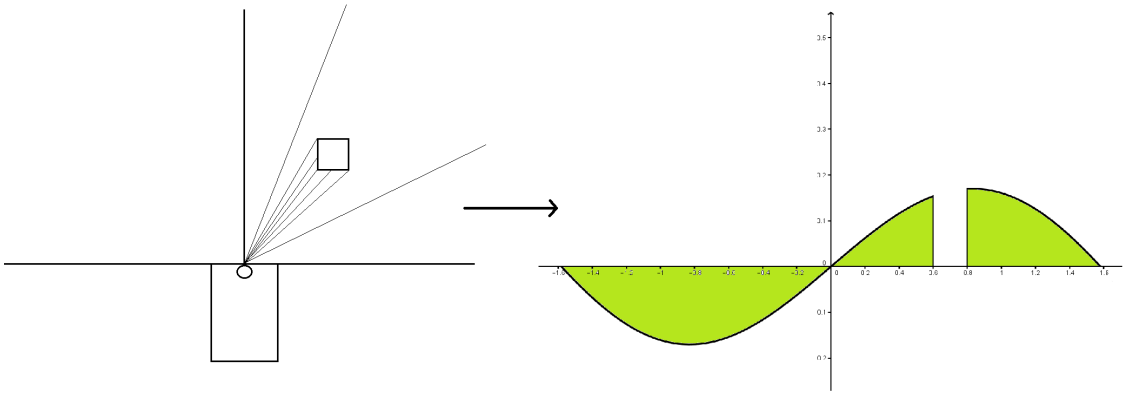


Abbildung 3.4.: Ein Hindernis taucht auf.

Um jedoch einen Ausweichwinkel zu errechnen, wird noch ein weiterer Wert benötigt.

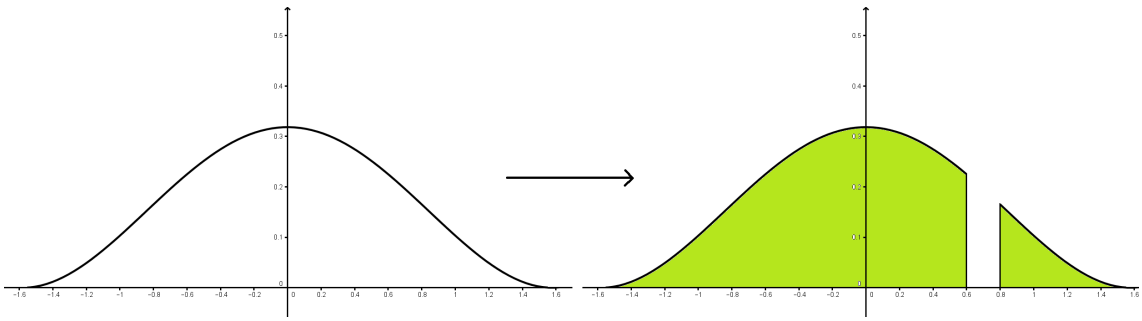


Abbildung 3.5.: $Y(\phi, x) = cos(\phi) * \mu_A(\phi) * \mu_B(x)$

Die eigentliche Defuzzyfizierung, also die Ableitung einer scharfen Stellgröße für die Fahrtrichtung wird nun durch die arctan-Funktion erreicht:
$\gamma = arctan(\frac{X}{Y}) = arctan(\frac{\sum_0^{180} sin(\phi)*\mu_A(\phi)*\mu_B(x)}{\sum_0^{180} cos(\phi)*\mu_A(\phi)*\mu_B(x)})$.

Der arctan gibt nun den neuen Fahrtrichtungswinkel Gamma zwischen der Ursprungsgeraden und der Ordinate (Y-Achse) zurück.

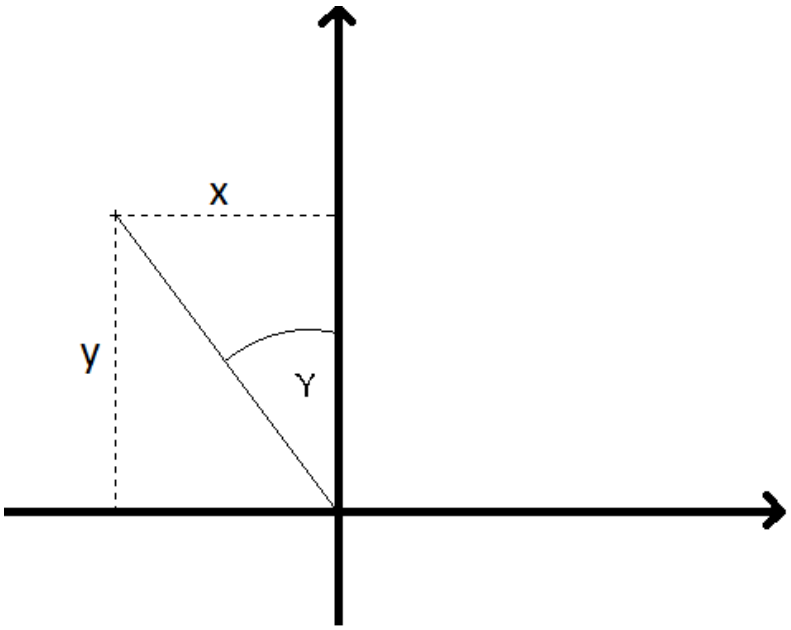


Abbildung 3.6.: atan2

3.1. Geschwindigkeitsregelung

An der oben beschriebenen Summenbildung der X- und Y- Funktionen wird der Nachteil des Verfahrens klar: Wenn auf beiden Seiten im gleichen Winkel und Abstand Hindernisse stehen, wird der Roboter nicht oder nicht weit genug ausweichen, da diesmal beide Seiten des Graphen "eingedellt" werden und die Gesamtsumme wieder nahe Null ist. Man merkt, dass der Y-Wert hier keinen Einfluss hat, da wenn der X-Wert nahe Null ist der Gesamtwinkel auch gegen Null geht. Um diesem Problem zu begegnen, wird die Geschwindigkeit je näher der Roboter einem Hindernis kommt fortwährend heruntergeregelt und schließlich auf Null gesetzt, um eine Kollision zu vermeiden. Ist der Roboter zum Stehen gekommen, wird seine Drehgeschwindigkeit auf einen festen Wert gesetzt, sodass sich der Roboter auf der Stelle dreht bis der Weg wieder frei ist.

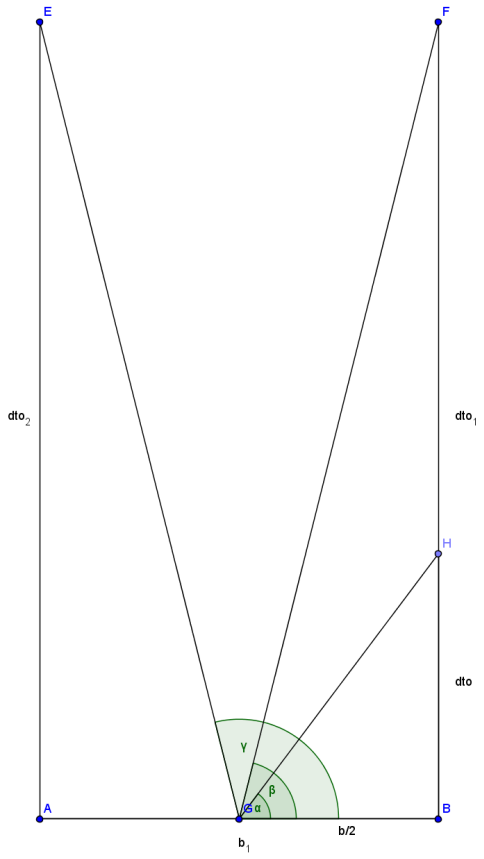


Abbildung 3.7.: virtuelle Straße

Um dies zu erreichen wird eine virtuelle Straße mit der Breite b des Roboters definiert. Wenn sich ein Hindernis H mit dem Abstand dto auf oder am Rand der Straße befindet, ist die Bedingung $|cos(\alpha)| * r < \frac{b}{2}$ erfüllt. Die Entfernung zu dem Hindernis in Fahrtrichtung ergibt sich durch:

$$dto = |sin(\alpha)| * r.$$

Aus der Beziehung $\frac{v}{v_{max}} = \frac{dto}{dto_{max}}$ kann die Geschwindigkeit als ein Quotient aus dto und einer Konstanten dto_max mal v_max beschrieben werden, die mit kleinerem dto immer kleiner wird. Wird dto kleiner als eine Konstante dto_min wird die Geschwindigkeit auf Null gesetzt und der Roboter dreht sich auf der Stelle. Bei der Drehrichtung wird natürlich die Tendenz des vorher berechneten Winkels γ mit einbezogen. Je nachdem, ob dieser positiv oder negativ ist, wird in die eine oder andere Richtung gedreht. Fälle, bei denen gedreht werden muss und nicht ausgewichen werden kann, sind trivialerweise vor allem Wände, seltener Hindernisse, die symmetrisch auf beiden Seiten des Roboters stehen.

4. GPS - Navigation

Die Steuerung mittels GPS und Kompasssensor ist eine überschaubare Sache, denn bei gegebenen Zielkoordinaten und den durch GPS-System gegebenen akutellen Koordinaten lässt sich damit immer ein Vektor zum Ziel ausrechnen. Mit Hilfe der Kompass-Nordrichtung kann ein Drehwinkel errechnet werden, um sich in Richtung Ziel zu drehen. Die einzige Schwierigkeit hierbei lag in der Angleichung der beiden unterschiedlichen Angaben für Winkel,

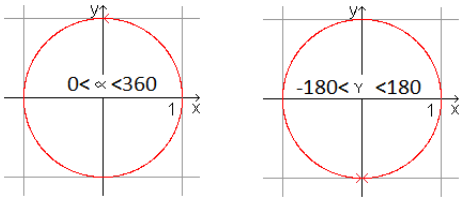


Abbildung 4.1.: Ziel- und Kompasswinkel

die beim Kompass von 0 bis 360° und bei dem Fahrtrichtungswinkel von -180 bis +180° lagen. Ich möchte dies hier anhand eines Beispiels exemplarisch berechnen. Zunächst gehen wir von einem Koordinatensystem aus, in dessen Ursprung sich der Roboter befindet und dessen Y-Achse die Nord-Süd-Richtung darstellt. Der vom Benutzer angegebene Zielpunkt muss wie folgt vom GPS Koordinatensystem in das Koordinatensystem des Roboters transformiert werden:

$$\vec{v}_{RBTdest} = \vec{v}_{GPSdest} - \vec{v}_{cur} = \begin{pmatrix} dest\_long \\ dest\_lat \end{pmatrix} - \begin{pmatrix} long \\ lat \end{pmatrix}$$

Danach kann mit dem arctan der Winkel γ zwischen der Ordinate und der Ursprungsachse durch den Zielpunkt berechnet werden. Der Winkel, der zurückgegeben wird, liegt zwischen +180° und -180°. Nun soll zwar die Ordinate die Nord-Süd Achse darstellen, allerdings ist der Roboter nur in den seltensten Fällen nach Norden ausgerichtet, deshalb muss mithilfe der Nordrichtung des Kompasses der Zielwinkel korrigiert werden. Da der Nordwinkel α jedoch einen Wert von 0 bis 360° hat, muss eine Fallunterscheidung für $\alpha + \gamma$ >180° vorgenommen werden.

$$\Phi(\alpha) = \begin{cases} für\ |(\alpha+\gamma)| < 180° : \gamma + \alpha \\ für\ |(\alpha+\gamma)| > 180° : \gamma + \alpha - 360 \end{cases}$$

Mithilfe dieser Berechnung erhält man einen Winkel, der im Koordinatensystem des Roboters unter Berücksichtigung seiner Ausrichtung in Richtung des Ziels zeigt.

```
1 void computeGpsDirection(arm_to_mb* am)
2 {
```

```
3     float dla,dlo,winkel1,tmp;
4     //Differenzen der Koordinaten bilden
5     dla = dest.latitude-(am->gps_data).latitude;
6     dlo = dest.longitude-(am->gps_data).longitude;
7
8     //ist das Ziel mit gewisser Toleranz erreicht?
9     if((fabs(dla*10000) < 3) && fabs((dlo*10000)) < 3)
10    {
11      reached = true;
12      printf("We have reached the destination.\n");
13    }
14    //korrespondierender Winkel fuer die GPS-Koordinaten
15    winkel1 = (atan2(dlo,dla)*180/M_PI);
16
17    tmp = winkel1 + (360-(float)(am->compass_data));
18
19    if(fabs(tmp) > 180)
20    {
21      tmp = fabs(tmp) - 360;
22    }
23    //printf("tmp: %f\n",tmp);
24    *direction2 = tmp;
25    return;
26  }
```

Listing 4.1: ComputeGpsDirection

Dies ist die Funktion, welche die GPS-Navigation und damit oben beschriebenes Verfahren implementiert.

5. Roboterkontrollarchitekturen

Spätestens an dieser Stelle muss die Frage aufkommen, in welcher Weise man diese beiden Navigationsmethoden in einen zeitlichen Ausführungskontext einbringt, der zur richtigen Zeit die richtige Methode anwendet. Um dies zu gewährleisten, gibt es mehrere Roboterkontrollarchitekturkonzepte, die jeweils eigene Vor- und Nachteile haben. Diese werden im Folgenden kurz vorgestellt.

5.1. Sequenziell vs Nebenläufigkeit

Zunächst sei gesagt, dass ein rein sequenzielles Roboterkontrollsystem nicht funktionieren kann. Denn sequenziell bedeutet, dass sich die Dauer eines Zyklus aus der Summe der Laufzeiten aller Module zusammensetzen würde, was bei potentiell sehr zeitaufwändigen Modulen, wie zum Beispiel der Wegplanung oder Objekterkennung, eine schnelle Reaktion auf Hindernisse oder generell auf irgend etwas unmöglich machen würde. Deshalb ist das sogenannte SMPA-Modell (engl. Sense Model Plan Act [3]) ein rein Theoretisches, um die Vorteile der sequenziellen Architektur zu verdeutlichen. Wie der Name bereits sagt, geht der Zyklus von Sensordaten (sense) aus, mithilfe derer ein Abbild der Umwelt "modelliert" werden kann. Aufgrund dieses Abbilds können die Aktionen geplant und schließlich ausgeführt werden. Ein Beispiel für einen Zyklus ist die Erstellung einer Karte mithilfe von Laserscannerdaten (sense and model), die Planung einer Route durch einen Suchalgorithmus wie den A*-Algorithmus und anschließendes Befahren dieser Route. Der Vorteil gegenüber einer nebenläufigen Struktur ist der, dass jedes Modul die Ergebnisse der Vorgänger mitbenutzen kann und somit Rechenaufwand spart, zum Beispiel durch Vermeidung von Mehrfachberechnungen.

Demgegenüber steht die Subsumptions- oder auch verhaltensbasierte Architektur. Dieser liegt zugrunde, dass sämtliche Sensordaten permanent durch echt oder konzeptionell nebenläufige Module abgefragt und verarbeitet werden. Der Vorteil gegenüber dem sequenziellen Modell liegt auf der Hand. Während ein Zyklus des letzteren wie gesagt eine Laufzeit von der Summe sämtlicher Module hat, ist hier ein Zyklus lediglich so lang wie das langsamste Modul. Diese Eigenschaft ermöglicht eine reaktionsschnelle Architektur, die durch verschiedene Verhaltensmodule den Roboter

steuert. So ist es möglich, dass ein Verhaltens-Modul zum Beispiel permanent die Laserscannerdaten mittels eines schnellen Algorithmus auf eventuelle Hindernisse überprüft um Kollisionen zu vermeiden, während ein anderes dieselben Daten benutzt, um daraus eine Karte zu bauen. Wenn es mehrere Module gibt, deren Ergebnis dieselben Stellgrößen des Roboters beeinflussen, kommt das Subsumptionsprinzip zum tragen, nach dem die Ergebnisse einiger Module die Ergebnisse anderer überschreiben, d.h. wenn der geplante Weg eine Fahrt nach links verlangt, seit dem letzten Update der Karte allerdings ein Hindernis in dieser Richtung aufgetreten ist, so wird in jedem Fall das Ergebnis "rechts" des Hindernis-Moduls beachtet werden.

Es ist Aufgabe des Programmierers zu den Modulen passende Subsumptionsbedingungen zu finden und zu implementieren. Im dem Fall unseres Roboters kann die Architektur folgendermaßen dargestellt werden.

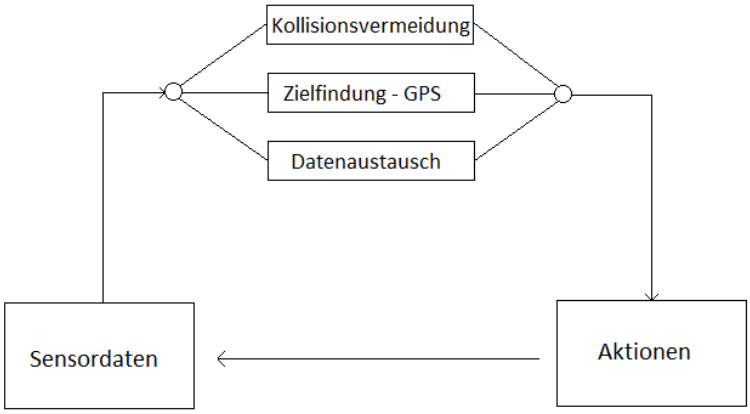


Abbildung 5.1.: Kontrollstruktur

Auch hier nimmt die Kollisionsvermeidung den höchsten Stellenwert ein und offensichtlich kann nur in Richtung Ziel gefahren werden, wenn der Weg weitestgehend frei ist. Dies ist genau dann der Fall, wenn der von der Freiraumnavigation berechnete Ausweichwinkel γ möglichst klein ist. Das ist die Subsumptionsbedingung für die beiden Navigationsmethoden. Der Datenaustausch zwischen dem Mainboard auf dem Roboter und dem ARM-Prozessor, und zwischen dem Mainboard un einem weiteren PC auf dem das GUI-Programm läuft, findet immer nebenläufig statt. Dies hat allerdings keinen Einfluss auf die Bewegungen des Roboters, weshalb hier das Subsumptionsprinzip nicht angewendet werden muss.

5.2. Implementation der Nebenläufigkeit

Ich möchte nun auf die Umsetzung der Nebenläufigkeit in C++ eingehen, da dies ein nicht unbeträchtlicher Teil des Lernprozesses im Zuge der besonderen Lernleistung gewesen ist. Zum größten Teil wird Nebenläufigkeit durch sogenannte Threads umgesetzt. Diese haben einen eigenen Stack und Programmzeiger, verhalten sich wie Prozesse, haben aber keinen eigenen Ausführungskontext, das heißt, sie teilen sich innerhalb eines Prozesses denselben Speicher. Dadurch können einfach Daten zwischen Threads ausgetauscht werden, da sie lediglich darauf zugreifen müssen, allerdings muss von mehreren Threads benutzter Speicher vor gleichzeitigem Zugriff geschützt werden um undefinierte Zustände zu vermeiden. In dem von uns genutzten Linux, gibt es die POSIX-Bibliotheken [5] die das Threading auf Low-Level Ebene regeln und folgende Funktionen zur Verfügung stellen.

Ein wichtiges Prinzip ist die sogenannte Mutual Exclusion (engl. gegenseitiges Ausschließen). Durch dieses Prinzip wird sichergestellt das jeweils nur ein Thread gleichzeitig Zugriff auf eine Ressource hat. In POSIX ist der gegenseitige Ausschluss durch eine Art Variable umgesetzt, genannt Mutex. Ein Thread kann einen Mutex "locken" (engl. einschließen), wodurch gesichert wird, dass nur dieser Thread Zugriff auf die dem Mutex zugeordnete Ressource hat, denn ein Mutex kann nur einmal "gelockt" werden, d.h. alle anderen Threads, die versuchen den Mutex zu locken, werden scheitern und an der entsprechenden Stelle in der Ausführung unterbrochen, bis der Thread, der den Mutex ursprünglich lockte, ihn wieder freigibt. Ein unterbrochener Thread verbraucht keine CPU-Ressourcen. Vom Programmierer muss sichergestellt werden, dass vor jedem Zugriff auf eine Ressource das Programm versucht den Mutex zu locken, es gibt demnach keine eingebaute Sicherheit.

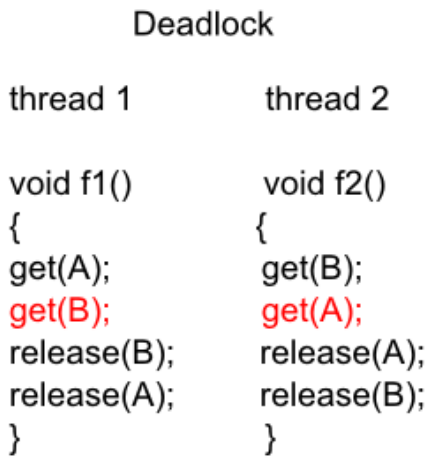


Abbildung 5.2.: Deadlock [2]

Eine weitere Herausforderungen an den Programmierer ist das Verhindern sogenannter Dead und Livelocks. Bei einem Deadlock versuchen zwei Threads zwei Mutexe zu locken, von denen einer vom jeweils anderen gehalten wird, das Programm kommt zum Stoppen, da beide die Ausführung unterbrechen, um auf den fehlenden Mutex zu warten. Dies geschieht jedoch nicht, da dieser im "Besitz" des jeweils anderen ist. Nebenstehend das Prinzip eines Deadlocks.

Das Livelock hingegen ist etwas komplizierter und lässt es sich folgendermaßen

beschreiben:

Zwei Threads versuchen zwei Ressourcen zu “locken“. Nur mit beiden können sie ihre Aufgabe durchführen. Zu einem Livelock kommt es nun, wenn bereits Maßnahmen zur Verhinderung eines Deadlocks genommen wurden. Das heißt, dass in diesem Beispiel die Threads sobald sie den anderen Mutex nicht locken können, ihren ursprünglichen Mutex freigeben. Im nächsten Zyklus der while-Schleife werden sie diesen jedoch erneut locken, was dazu führt, dass niemals einer der beiden Threads beide Mutexe halten kann. Dennoch wird die Ausführung nicht unterbrochen, da die Threads damit beschäftigt sind Mutexe zu locken und wieder freizugeben.

Livelock

```
thread 1                thread 2

void f1()               void f2()
{                       {
 int got_it=0;           int got_it=0;
 while(!got_it) {        while(!got_it) {
  get(A);                 get(B);
  if(acquired(B)) {       if(acquired(A)) {
   release(B);             release(A);
   release(A);             release(B);
   got_it=1;               got_it=1;
  }                       }
  else {                  else {
   release(A);             release(B);
  }                       }
 }                       }
}                       }
```

Abbildung 5.3.: Livelock [2]

Weiterhin muss die Condition Variable oder bedingte Variable erwähnt werden, welche ermöglicht, dass Threads sich untereinander signalisieren. Es ist eine komplexere Art der Synchronisation, als jene, die von Mutexen ermöglicht wird. Die Condition Variable erlaubt es Threads auf ein bestimmtes Ereignis zu warten ohne CPU-Ressourcen zu verwenden. “Trifft“ ein Thread in seiner Ausführung auf eine bedingt Variable, so wird seine Ausführung unterbrochen, bis dieser bedingten Variable von einem anderen Thread signalisiert wird. In meinem Programm verwende ich eine bedingte Variable, um dem Hauptthread zu signalisieren, dass neue Laserscannerdaten vorhanden sind.

Als letztes Instrument der POSIX-Bibliotheken sei der Thread-Join genannt. Dies ist eine Funktion, die es einem Thread erlaubt auf die Termination der Ausführung eines anderen Threads zu warten. In meiner Implementation ist das an einer Stelle nötig. Wenn der Hauptthread, und damit das gesamte Programm, vor dem Thread der den Laserscanner ausliest terminiert, wird das USB-Interface, des Laserscanners nicht freigegeben.Der PC muss dann neu gestartet werden, damit das Programm erneut ausgeführt werden kann, da es sonst zu Fehlermeldungen kommt.

6. Implementation

Nun folgt eine etwas detailliertere Beschreibung meiner Programmstruktur.

Die Programme wurden in C++ mit der kostenlosen Entwicklungsumgebung Eclipse auf der Linux Distribution Debian geschrieben.

Insgesamt haben zwei Threads ausgereicht, um sämtliche Funktionen umzusetzen. In einem wird der Laserscanner permanent ausgelesen und eine Ausweichrichtung berechnet, während zum anderen dem Hauptthread, die Kommunikation zu und von ARM und Pc stattfindet, sowie die GPS Orientierung und die Überprüfung der Susumptionsbedingung. Die Verarbeitung der GPS Daten in diesem Thread bietet sich an, da sie dort direkt vom ARM Prozessor empfangen werden. Die Richtungsdaten der Freiraumnavigation werden über den Heap ausgetauscht. Ein Speicherbereich, auf den beide Threads zugreifen können, wobei jedesmal wenn eine neue Richtung berechnet worden ist, der Hauptthread durch eine Condition Variable signalisiert wird. Das heißt auch, dass der Thread bis dahin still steht und somit nur mit der Frequenz des Laserscanner von 15 Hz “aufgeweckt“ wird. Der Vorteil wäre theoretisch, dass der Nebenthread, sobald er dem Hauptthread signalisiert hat, sofort neue Laserscannerdaten empfangen kann. In der Regel dauert die Übertragung per TCP/IP allerdings nur sehr kurz und der Thread wartet die meiste Zeit auf das Eintreffen neuer Daten.

```
27 //auf einen neuen Scan warten
28     rc = pthread_mutex_lock(&mutex1);
29     rc = pthread_cond_wait(&new_scan,&mutex1);
30     dir = *direction1;
31     spd = *speed;       //werden von zwei Threads benutzt
32     rc = pthread_mutex_unlock(&mutex1);
```

Listing 6.1: Condition Variable

Dies ist besagte Stelle an der gewartet wird. Dazu wird zuerst der Mutex "gelockt". Durch Aufrufen von "pthread_cond_wait" wird der Mutex zunächst wieder freigegeben bis er automatisch, nachdem dem Thread signalisiert worden ist wieder "gelockt" wird. Man merkt, dass die Variablen “dir“ und “spd“ tatsächlich die einzigen beiden Variablen sind, auf die beide Threads zugreifen und die deshalb durch einen Mutex geschützt werden müssen.

6.1. TCP/IP

Das zur Kommunikation zwischen ARM-Prozessor und Computer verwendete TCP\IP wurde zumindest im Bereich der C++ Implementation neu erlernt und angewendet. Aus Lerngründen wurden nur die Linux Bibliotheken verwendet, was das Schreiben des Socket-Handling erfordert. Dies habe ich in einer eigenen Bibliothek, namens "networkFactory"realisiert. In der aktuellen Implementation, läuft auf dem Mainboard ein Server, zu dem sich die GUI als Client verbinden kann. Bei der Verbindung zum ARM dient dieser als Server und das Mainboard kann sich als Client verbinden.

```
33 class networkFactory {
34 public:
35   networkFactory();
36   virtual ~networkFactory();
37   int open_client(char *IP4,char *Port,int options);
38   int create_host(char *Port);
39 private:
40   struct addrinfo hints, *serverinfo,*i;
41   int sockfd,newsockfd,status;
42 };
```

Listing 6.2: class NetworkFactory

Die Bibliothek “networkFactory“ stellt zwei Funktionen zur Verfügung. Erstens die Funktion open_client welche sich zu dem übergebenen Port und IP Adresse verbindet und einen entsprechenden File-Deskriptor zurückgibt, der dem Socket entspricht. Innerhalb der Funktion werden alle nötigen System Calls ausgeführt, darunter vor allem die Funktionen “socket“, welche einen natürlich einen Socket erstellt und die Funktion “connect“, welche den Socket zu der gegebenen Adresse verbindet. Der erstellte Socket ist normalerweise im blockierenden Modus, dass heißt, das Funktionen wie “connect“, bei einem Verbindungsversuch warten werden bis sie sich verbinden können, oder bis nach 20 Sekunden ein Timeout erfolgt. Dies war für unsere Anwendung eine zu lange “Timeout-Zeit“. Der Socket muss daher auf Non-Blocking umgestellt werden. Dies bedeutet, dass zum Beispiel “connect“ sofort einen Timeout hat und den Error EINPROGRESS zurückgibt. Nun kann mit Hilfe einer weiteren Funktion namens “select“ solange wie gewünscht gewartet werden. So kann eine manuelle Timeout-Zeit eingestellt werden.

Die Funktion create_host erstellt einen Socket und bindet ihn an einen bestimmnten Port, wodurch dieser Socket einen TCP\IP Server darstellt. Auch hier wird innerhalb der Funktion die Funktion “socket“ genutzt um einen Socket zu erstellen. Des Weiteren wird die Funktion “bind“ benutzt, um den Socket an einen Port zu binden. Diese Funktion verbidnet sich nicht mit einem anderen PC, weshalb sie nicht blockierend ist.

7. GUI

Die GUI ist vor allem zum Überprüfen von Roboter Parametern und Daten, da das mit einer GUI wesentlich komfortabler ist als mit einer Konsolenausgabe. Vor allem die graphische Darstellung der Messpunkte des Laserscanners ist nicht nur schön anzusehen, sondern ermöglicht auch die Vorhersage des Verhaltens des Roboters und anschließendes Verifizieren. So kann zum Beispiel überprüft werden auf welche Weise ein Hindernis in dem 2D Scan auftaucht, woraus manchmal eine fehlende oder unerwartete Reaktion auf ein Hindernis erklärt werden kann.

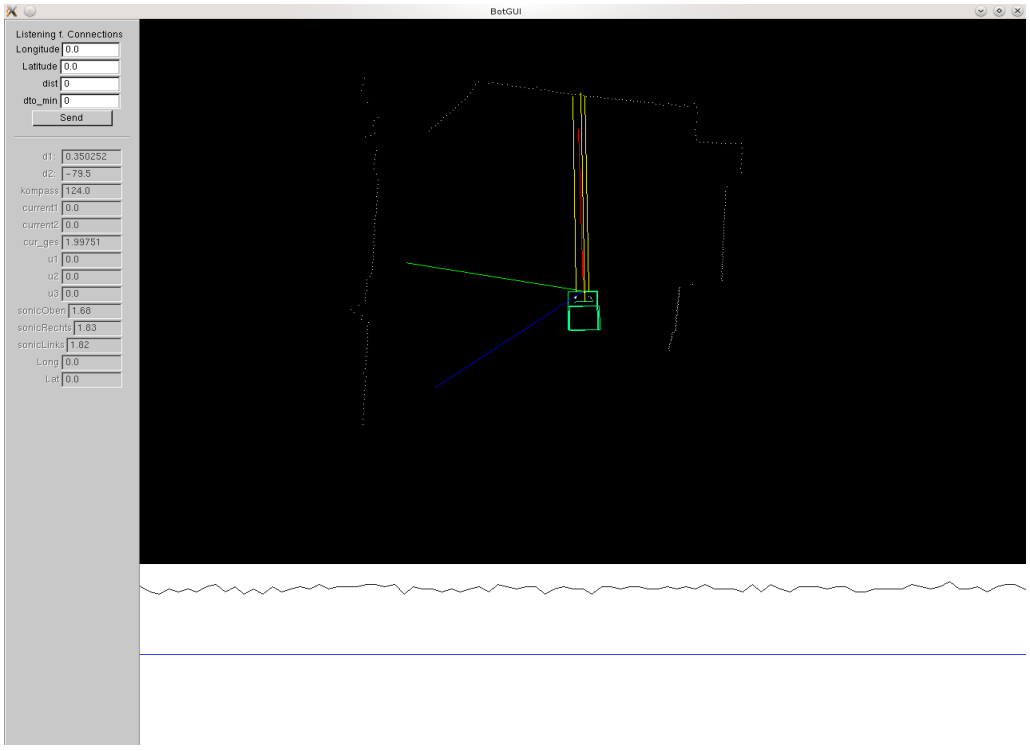


Abbildung 7.1.: GUI

Die Gui ist in C++ geschrieben und benutzt OpenGl zur Darstellung, sowie die Glut Bibliotheken für das Fenstermanagement und die Glui Bibliothek für die UI Elemente wie Button und Textfeld.

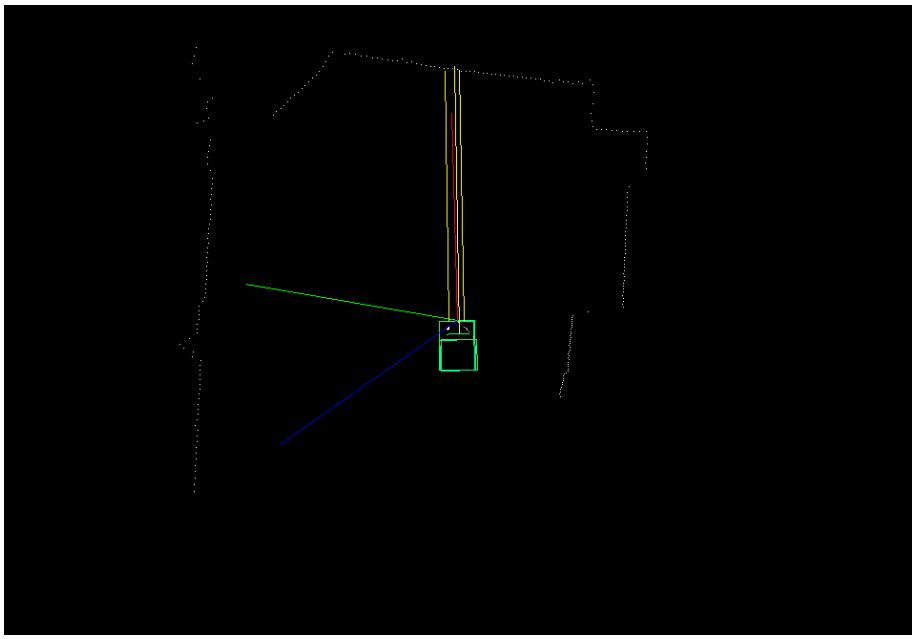


Abbildung 7.2.: Darstellung der Laserscannerdaten

Der rote Strich ist der Freiraumnavigation zugeordnet und zeigt in die aktuelle Ausweichrichtung, während der blaue Strich nach Norden zeigt. Weiterhin korrespondieren die drei gelben Striche mit den Werten der Ultraschallsensoren und der Grüne zeigt in Richtung der angegebenen GPS-Koordinaten. Es ist anzumerken, dass sich nicht die Darstellung des Roboters in der Umgebung bewegt sondern die Umgebung sich um den Roboter bewegt, da Ersteres in der Animation zu kompliziert ist.

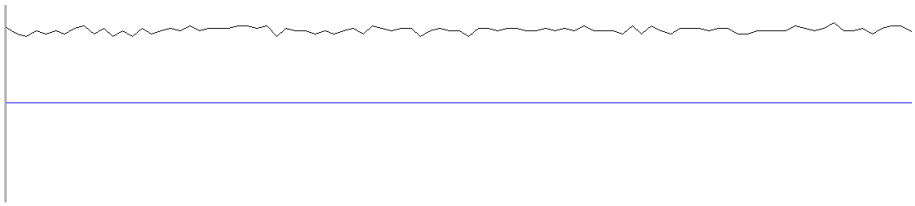


Abbildung 7.3.: Strom Graph

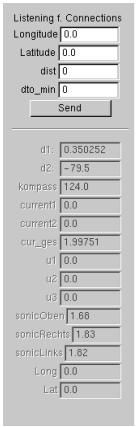


Abbildung 7.4.: Ausgabe der Sen-

In Abbildung 7.3 schwarz dargestellt ist der Strom der von Mainboard, und ARM-Prozessor samt Peripherie verbraucht wird. Blau dargestellt ist der Strom der von den sechs Motoren verbraucht wird, allerdings fährt der Roboter zum Zeitpunkt der Aufnahme dieses Bildes nicht, daher fließt kein Strom.

Und zu guter Letzt ist da die Werte- und Parameterleiste. Sie zeigt einige Werte an, wie z.B. die aktuelle GPS Position, sowie die Werte sämtlicher Sensoren. Außerdem wird es möglich sein einige Paramter des Roboters wie die Zielkoordinate über die GUI zu ändern.

8. Fazit

Das Ziel der Besonderen Lernleistung einen autonomen Roboter, der Hindernissen ausweicht und ein Ziel sucht, ist erreicht. Allerdings ist die Arbeit an dem Roboter noch nicht abgeschlossen, da weitere Projekte damit umgesetzt werden können. Somit, ist dies lediglich ein Zwischenbericht. Insgesamt lässt sich sagen, dass diese Besondere Lernleistung eine lehrreiche und herausfordernde Arbeit war. Die gewonnene Erfahrung im Bereich des Programmierens und Durchführen eines Projekts in Teamarbeit wird sicher auch später noch sehr nützlich sein. Natürlich hatte ich auch eine Menge Spaß bei der Durchführung der Lernleistung. In Anbetracht der Tatsache, dass die autonome Navigation von Robotern zunehmend an Bedeutung gewinnt, ist das Erwerben von Wissen in dieser Richtung hoch anzusehen. Dazu kommt die Vertiefung meines Wissen in dem Textsatzsystem LaTeX, das beim Schreiben von weiteren umfangreicheren Arbeiten nützlich sein wird.

Literaturverzeichnis

[1] Cooksey, Diana: *Understanding the Global Positioning System (GPS)*. http://www.montana.edu/gps/understd.html, . – [Online; accessed 28-Jan-2013]

[2] bogotobogo: *Multithreading*. http://www.bogotobogo.com/cplusplus/multithreaded.php, . – [Online; accessed 28-Jan-2013]

[3] Hertzberg, Joachim ; LingeMann, Kai ; Nüchter, Andreas: *Mobile Roboter*. Springer Vieweg, 2012

[4] Wikipedia: *Fuzzy-Regler*. http://de.wikipedia.org/wiki/Fuzzy-Regler, . – [Online; accessed 28-Jan-2013]

[5] N.a: *Multi-Threaded Programming With POSIX Threads*. users.actcom.co.il/~choo/lupg/tutorials/multi-thread/multi-thread.html#thread_mutex_whatis, . – [Online; accessed 28-Jan-2013]

[6] Hall, Brian: *Beej's Guide to Network Programming*. http://beej.us/guide/bgnet/output/html/multipage/index.html, . – [Online; accessed 28-Jan-2013]

[7] Loco: *Connect with timeout (or another use for select()*. http://developerweb.net/viewtopic.php?id=3196, . – [Online; accessed 28-Jan-2013]

A. Anhang

A.1. Inhalt der CD

- Sourcen - In diesem Ordner befinden sich die Sourcen.
- Webseiten - Hier sind alle benutzten Webseiten gespeichert.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Ort, Datum	Mark Springer