Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Robotik

Untersuchung der Effizienz von RRT* bei autonomen Autos

Bernd Sahre
Matrikelnummer: 4866892
besahre@zedat.fu-berlin.de

Betreuer:	Prof. Dr. Daniel Göhring
Eingereicht bei:	Prof. Dr. Daniel Göhring
Zweitgutachter:	Prof. Dr. Raul Rojas

Berlin, 8. Mai 2018

Zusammenfassung

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

8. Mai 2018

Bernd Sahre

Inhaltsverzeichnis

1 Einleitung

Der Traum vom autonomen Fahren reicht bis in die Antike zu den Griechen zurück, deren Gott des Feuers und der Schmiedekunst Hephaistos Androiden und selbst fahrende Automobile konstruierte. Mittlerweile ist die Forschung in diesen Bereichen von der Phantasie in die Wirklichkeit gerückt und so weit fortgeschritten, dass dieser Traum schon bald Wirklichkeit werden könnte. Zuvor müssen jedoch etliche Herausforderungen bewältigt werden. Eine davon ist, das Auto sicher durch den Straßenverkehr zu bringen. Dabei heißt sicher, dass das Auto während der Fahrt weder sich noch andere Verkehrsteilnehmer gefährdet. Neben der Sicherheit ist jedoch auch das möglichst schnelle Erreichen des Ziels wichtig, bei dem das Auto seinen Weg durch eine Umgebung mit statischen und dynamischen, das heißt sich bewegenden Hindernissen finden muss. In diesem Kontext ist noch zu erwähnen, dass dieser Weg nicht wie bei einem Navigationssystem beispielsweise von Berlin nach Hamburg führt, sondern eher von einer Straßenecke zur nächsten oder über einen Parkplatz. Dies wird durch einen Pfadplaner gewährleistet, der -informell gesprochen - aus der eigenen Position, dem Zielbereich und unter Berücksichtigung aller statischen und sich bewegenden Hindernissen einen sicheren Pfad zum Ziel findet. Dieser Pfad ist dann durch das Auto abfahrbar.
Um diese Aufgabe zu meistern, existieren unterschiedliche Ansätze, um dem Auto je nach Anwendungsfall bei gegebenen Ziel eine *Trajektorie* vorzuschlagen. Die jeweiligen Algorithmen unterscheiden sich in Ausführungszeit, Genauigkeit, Sicherheit und berechnen unterschiedlich optimale Pfade.
Das Dahlem Center for Machine Learning and Robotics untersucht maschinelles Lernen und Anwendungen intelligenter Systeme. Dazu haben sich vier Arbeitsgruppen der Freien Universität Berlin zusammengeschlossen:

- Intelligent Systems and Robotics (Prof. Dr. Raúl Rojas)
- Autonomos Cars (Prof. Dr. Daniel Göhring)
- Artifical and Collective Intelligence (Prof. Dr. Tim Landgraf)
- Logic and automatic proofs (Christoph Benzmüller)

Ein Forschungsgebiet ist die Entwicklung und Analyse autonomer Autos. Zur oben beschriebenen Pfadplanung wird in der Arbeitsgruppe hauptsächlich das Prinzip elastischer Bänder (Time-Elastic-Bands, [vgl. 4]) zur Erzeugung abfahrbarer *Trajektorien* benutzt. Doch auch die Untersuchung und Analyse anderer Algorithmen ist interessant, um zu überprüfen, ob sich eine vertiefte Forschung in diesen Bereichen lohnt.
Diese Bachelorarbeit untersucht einen dieser anderen Algorithmen zur Pfadplanung, *RRT**, auf seine Tauglichkeit, über eine vorgegebene Fahrbahn mit Hindernissen eine abfahrbare, sichere und möglichst optimale *Trajektorie* zu finden.

1.1 Aufbau und Struktur

Nach einer kurzen Hinführung zum Thema werden zuallererst die Grundlagen (2) besprochen, um das Verständnis der nachfolgenden Kapitel zu erleichtern. Danach

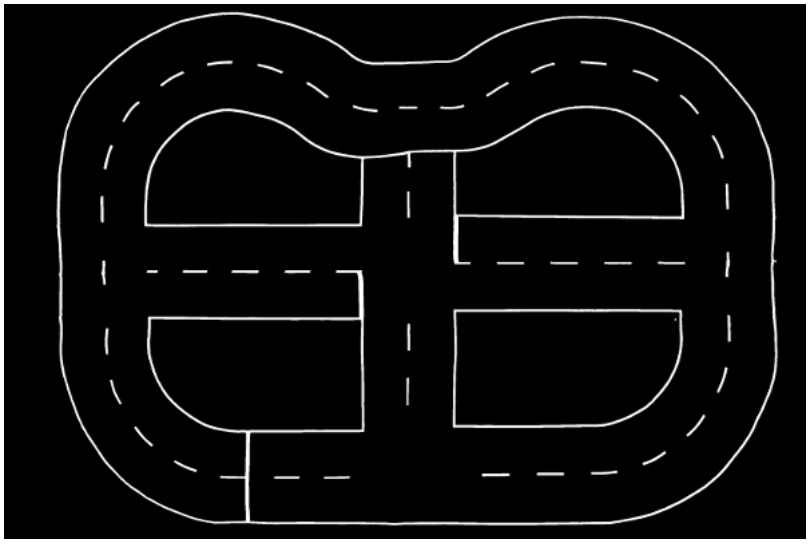

Abbildung 1: Das Auto fährt auf der äußeren Strecke einmal im Kreis

wird die Umsetzung des Algorithmus auf das Auto sowie die technischen Details und Hintergründe beschrieben (3). Im vierten Kapitel (**??**) werden die Ergebnisse der Testfahrten vorgestellt und bewertet. Zum Schluss (5) werden in einem Fazit alle Schlussfolgerungen nochmals zusammengefasst und ein Ausblick auf weitere mögliche Forschungsmöglichkeiten gegeben.

1.1.1 Problemanalyse

Diese Arbeit untersucht die Effizienz des *RRT**-Algorithmus unter Anwendung bei autonomen Autos. Dazu fährt das Auto eine vorgegebene Strecke ab. Auf dieser Strecke werden erst statische, dann sich bewegende Hindernisse platziert.
Das Auto soll diese Strecke in möglichst kurzer Zeit mit einem möglichst optimalen Pfad abfahren, ohne eines dieser Hindernisse zu berühren. Dazu sollte das Auto den Algorithmus idealerweise 30 Mal pro Sekunde ausführen, mindestens aber vier Mal pro Sekunde. Dadurch wird gewährleistet, dass der entstehende Pfad fürs Auto sich an bewegende Hindernisse anpasst.
Die Effizienz, Sicherheit und Effektivität des Algorithmus wird bei verschiedenen Eingabeparametern untersucht und bewertet.

1.1.2 Anmerkung zur Gestaltung der Arbeit

[TODO Plagiat mit Prof abklären (ist ne 1:1 Kopie)] Für die im Folgenden verwendeten personenbezogenen Ausdrücke wurde, um die Lesbarkeit der Arbeit zu erhöhen, ausschließlich die männliche Schreibweise gewählt. Desweiteren werden eine Reihe von englischen Bezeichnungen und Fachwörtern verwendet, um einerseits dem interessierten Leser das Studium der fast ausschließlich englischen Fachliteratur zu erleichtern und andererseits bestehende Fachbegriffe nicht durch die Übersetzung zu

verfälschen. Bei diesen Begriffen wird zur Unterscheidung eine kursive Schriftart verwendet. [TODO: Wenn möglich, wurden die Quellen angegeben, bei den anderen wurde zur Definition Wikipedia verwendet.]

1.1.3 Glossar

Begriff	Erklärung
Trajektorie	Die Strecke, die dem Low-Level-Planer des Autos übergeben wird
Low-Level-Planer	Steuert direkt die Motoren des Autos, Lenkung und Antrieb, um eine vorgegebene Trajektorie möglichst genau abzufahren.
RRT	Rapidly-Exploring Random Tree [9], ein Algorithmus zur Findung eines Pfades zum Ziel durch unbekanntes Gelände, wird in Kapitel 2.1 noch weiter erläutert
RRT*	Eine verbesserte Variante des RRT, bei dem die Pfade optimiert werden [11]. Asymptoptisch optimal, erläutert in Kapitel 2.2
ROS	Robot Operating Systems, eine Open-Source Sammlung an Software-Bibliotheken und Werkzeugen zur Kreation von Anwendungen zur Robotik [6].
Nonholonomic Robots	Roboter, die gewissen kinematischen Einschränkungen unterworfen sind. Ein Auto zum Beispiel kann sich nicht in jede beliebige Richtung bewegen, sondern ist z.B. durch den maximalen Lenkwinkel und den Wenderadius eingeschränkt und kann nicht jeden beliebigen Punkt sofort, mit nur einem Schritt, erreichen. Im Gegensatz dazu stehen holonome Roboter, die sich in jede beliebige Richtung ohne direkte Einschränkungen bewegen können.
Kinodynamic planning	Beschreibt eine Klasse von Problemen, bei der physikalische Einschränkungen wie Geschwindigkeit, Beschleunigung und Kräfte zusammen mit kinematischen Einschränkungen (z.B. Hindernisvermeidung) berücksichtigt werden müssen.
hochdimensionale Probleme	Probleme, bei deren Lösung nicht nur ein oder zwei, sondern sehr viele verschiedene Parameter berücksichtig werden müssen. Kinodynamic Planning befasst sich mit hochdimensionalen Problemen.
randomisierte Algorithmen	Algorithmen, deren Durchführung nicht determiniert ist, die also jedesmal ein etwas anderes Ergebnis zurückliefern. Der Vorteil ist, dass dies Algorithmen zwar nicht immer das bestmöglichste Ergebnis liefern, dafür aber schneller in der Ausführungszeit sind und oft einfacher zu verstehen und zu implementieren.

Begriff	Erklärung
Odroid	Einplatinencomputer am Auto mit Mehrkernprozessor.
Arduino Nano	Mikrocontroller am Auto zur Steuerung der Motoren und Sensoren.
Gyroskop	Auch Kreiselinstrument, Sensor am Auto zur Messung der Lageänderung (z.B. Ausrichtung des Autos).
Lidar	Light detection and ranging; Radarscanner am Auto, für Abstandsmessung zu Hindernissen.
Odometrie	Methode zur Schätzung von Position und Orientierung anhand der Daten des Vortriebsystems (Motoren).
Rewiring	Neuverknüpfung eines RRT*-Baumes [11]. Die Begriffe Rewiring und Neuverknüpfung werden synomym gebraucht. Nach Hinzufügung eines Knotens K zum Baum werden Nachbarknoten überprüft, ob diese durch K besser erreichbar sind als vorher. Falls ja, wird K als Vaterknoten ausgewählt.
k-nearest neighbour algorithm	Algorithmus, der zu einem Punkt P die nächsten K Nachbarn bestimmt.
radius k-nearest neighbour	Algorithmus, der in einem Radius alle nächsten Nachbarn des Punktes P bestimmt.
Quaternion	4-dimensionaler Vektor, in dem die Ausrichtung eines Objekts im Raum definiert ist.

1.1.4 Wichtige Quellen und deren Beitrag zum Thema

Es existieren[TODO RRT-Quellen [vgl 9], RRT*-Quellen, Übersichtsliteratur]
Nun werden wir uns den wissenschaftlichen Grundlagen der Arbeit widmen.

2 Grundlagen

Dieses Kapitel führt die verwendeten Algorithmen und Berechnungen ein.
Mit dem A*-Algorithmus wurde schon in den 60er Jahren ein Werkzeug für die Pfadplannung von Robotern eingeführt [**?**]. Pfadplannung bedeutet hier, dass ein Roboter mit festgelegten kinematischen Einschränkungen in einer bestimmten, definierten Umgebung von einem Startzustand zu einem Zielzustand mithilfe von Steuerungseingaben gelangen kann, ohne die physikalischen Gesetze und die der Umgebung (keine Kollision mit Hindernissen) zu verletzen. Der A*-Algorithmus löst dieses Problem mit vielen Einschränkungen, indem er in einem Graphen den kürzesten Weg zwischen zwei Knoten findet. Allerdings benötigt A* diesen Graphen zur Berechnung des Weges und ist aufgrund des hohen Speicherplatzbedürfnisses für *hochdimensionale Probleme*, d.h. für Probleme mit den oben genannten Einschränkungen, nicht geeignet [**?**].

In nachfolgender Zeit wurden *randomisierte Algorithmen* entwickelt, die zwar nicht die mathematisch optimalste Lösung lieferten, dafür bedeutende Geschwindigkeitsvorteile hatten. da in der Praxis oft viele Faktoren aufgrund der hohen Komplexität bei geringem Einfluss gar nicht berücksichtig werden, reicht es, nur bis zu einem gewissen Grade die exakte Lösung zu liefern.
Diese Ansätze wurden vom *randomized potential field* Algorithmus [8] und dem *probalistic roadmap* Algorithmus [1] verfolgt. Doch auch diese waren nicht allgemein auf *nonholonomic Robots* anwendbar und lösten oft nur spezifische Probleme unter ganz bestimmten Bedingungen. Der Erfolg des *randomized potential field* Algorithmus beispielsweise hing stark von der Wahl einer passenden Heuristik ab [vgl. Kap 3.4 in 8]. Während sich bei einfachen Ausgangsbedingungen die Heuristik noch einfach finden lies, wurde dies bei komplexen, dynamischen Umgebungen mit Hindernissen, physikalischen und kinematischen Bedingungen zu einer großen Herausforderung.
1998 schließlich führte Steven LaValle den *RRT*-Algorithmus ein, der die oben genannten Einschränkungen umgehen sollte.

2.1 RRT

LaValle erkannte die die Vorteile von randomisierten Algorithmen und die ërfolgreiche Einsetzung im generellen Problem der Pfadplanung". [Kap 1 9]. Jedoch sah er auch die Einschränkungen der vorhandenen randomisierten Algorithmen. Insbesondere störte ihn die fehlende Skalierbarkeit in komplexeren und hochdimensionalen Umgebungen, da diese Algorithmen damit nur unter gewissen Vorbedingungen effizient einsetzbar waren. Der *probalistic roadmap* Algorithmus [1] beispielsweise beinhaltet einen "lokalen Planer", der zwar für holonomische Systeme und Roboter effiziente Ergebnisse liefert, aber bei nicht holonomischen Fahrzeugen und auf allgemeine Probleme angewandt schwindet die Effizienz der *probalistic roadmap* [9] .
Bei der Entwicklung von *RRT* wurde deshalb viel Wert auf Einfachheit, Allgemeingültigkeit und damit auf Skalierbarkeit gelegt [vlg. Kap 3 9]. Bevor wir jedoch genauer auf die Vorteile des Algorithmus eingehen und warum dieser hier gewählt wurde, folgt jetzt erstmal eine kurze Erklärung der Funktionsweise. RRT baut einen Baum auf, indem zufällig gewählte Punkte unter Berücksichtigung einer Metrik verbunden werden. Der Algorithmus mit dem *RRT* T mit den Eingabeparametern Größe K, Metrik M, Bewegungsfunktion u, und Startzustand `x_init` funktioniert folgendermaßen:

2.1.1 Funktionsweise RRT

[TODO in Quellcode Literaturverweis]

```
1 BUILD_RRT(K, M, u, x_init)
2   T.init (x_init)
3   for k=1 to K do
4     x_rand = RANDOM_STATE();
5     EXTEND(T, x_rand);
6   Return T;
```

```
1 EXTEND(T, x)
2   x_near = NEAREST_NEIGHBOR(x, T, M);
```

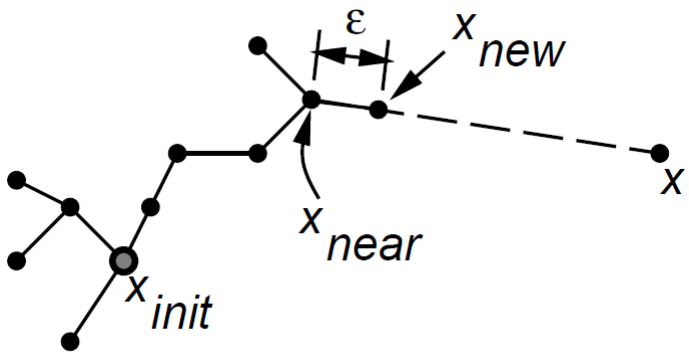

Abbildung 2: Die EXTEND Funktion [10]

```
3   x_new = project(x, x_near, u);
4   if (Collisionfree(x_new, x_near, u) then
5     T.add_vertex(x_new);
6     T.add_egde(x_near, x_new, u_new);
7     Return Extended;
8   else
9     Return Trapped;
```

Der Baum wird anfangs mit dem Startzustand `x_init` initialisiert. Anschließend wird in K Iterationen der Baum T aufgebaut, indem mit `x_rand` ein zufälliger Punkt ausgewählt und mit `EXTEND(T, x_rand)` dem Baum hinzugefügt wird.
Die Funktion `EXTEND(T, x)` ermittelt zunächst mithilfe der Metrik M den nächsten Nachbar von x. Diese Metrik kann von einer einfachen euklidischen Distanz bis hin zur komplexen Einberechnung verschiedener kinetmatischer Bedingungen alles beinhalten. Die hier verwendete Metrik wird später noch in Kapitel 3.6 beleuchtet.
Ist der nächste Nachbar `x_near` gefunden, wird von diesem aus mit `project(x, x_near, u)` ein Schritt der Länge ϵ in Richtung x durchgeführt und an dieser Stelle der neue Knoten `x_new` erzeugt.
Nun wird mit `Collsionfree(x_new, x_near, u)` überprüft, ob `x_new` oder die Bewegung u zu `x_new` hin mit Hindernissen kollidiert oder diesen zu Nahe kommt. Falls dies nicht der Fall ist, werden sowohl der neu entstandene Knoten `x_new` als auch die Kante von `x_near` zu `x_new` dem Baum T hinzugefügt.
Falls `x_new` oder die Bewegung u zu `x_new` mit Hindernissen kollidiert oder diesen zu Nahe kommt, wird der Knoten `x_new` verworfen und die Funktion `EXTEND(T,x)` beendet.

2.1.2 Vor- und Nachteile von RRT

Die Rapidly-Exploring Random Trees haben einige Eigenschaften, die für Bewegungsplanung von Robotern von großem Vorteil sind, wie LaValle in [Kapitel 3 in 9] schreibt:

1. Ein *RRT* breitet sich sehr schnell in unerforschte Bereiche des Statusraums aus. Dadurch können Pfade schnell gefunden werden und es wird schnell eine mög-

liche (wenn auch nicht optimale) Lösung gefunden.

2. Die Verteilung der Knoten im Baum entspricht der Verteilung, wie diese Knoten erzeugt wurden; dies führt zu konsistentem Verhalten. Unter anderem kann dadurch das Wachstum des Baumes in eine bestimmte Richtung gesteuert werden (z.B. zum Ziel hin)

3. Ein *RRT* ist probabilistisch vollständig, das heißt mit zunehmender Laufzeit konvergiert die Wahrscheinlichkeit, keinen Pfad zum Ziel zu finden, gegen null

4. Ein *RRT* ist sowohl einfach zu implementieren als auch einfach in der Analyse, was es ermöglicht die Performance einfach zu analysieren und zu verbessern

5. Ein *RRT* ist immer mit sich selbst verbunden, und das bei einer minimalen Kantenanzahl

6. Ein *RRT* kann als Pfadplanungsmodul interpretiert werden, was die Kombination mit anderen Werkzeugen zur Bewegungsplanung möglich macht

Leider existieren neben den oben genannten Vorteilen auch etliche Nachteile. Eines der größten ist, dass ein *RRT* nicht den optimalen Pfad zurückliefert, da einmal gesetzte Knoten ihren Vaterknoten nicht mehr ändern können. Dadurch kann, auch wenn durch Neuverknüpfung eine bessere Knotenfolge vom Start zum Ziel bestehen würde, diese nicht erstellt werden.

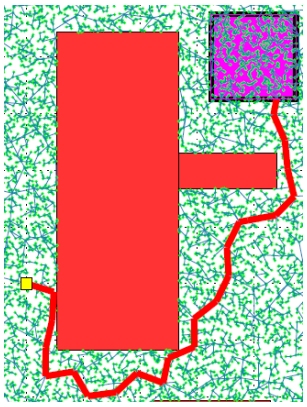

Abbildung 3: RRT bei 20.000 Knoten

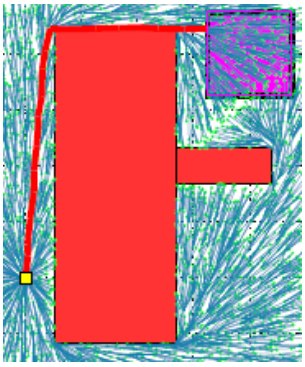

Abbildung 4: RRT* bei 20.000 Knoten

Deshalb wurde von Sertac Karaman und Emilio Frazzoli aufbauend auf *RRTs* der Algorithmus *RRT** eingeführt, welcher diesen Nachteil ausgleicht.

2.2 RRT*

Im Gegensatz zu einem *RRT* führt ein *RRT** die zwei folgenden Neuerungen ein:

1. Auswahl eines passenden Vaterknotens bei Hinzufügen des Knotens zum Baum
2. Neuverknüpfung des Baumes

Diese Neuerungen resultieren in einer veränderten `EXTEND(T,x)` Funktion, siehe [11].

2.2.1 Funktionsweise RRT*

```
1  EXTEND(T,x)
2    x_nearest = NEAREST_NEIGHBORS(x, T, M);
3    x_new = project(x, x_nearest, u);
4    if (Collisionfree(x_new, x_near, u) then
5      T.add_vertex(x_new);
6      x_min= x_nearest;
7      c_min = x_nearest.cost + cost(x_nearest, x_new);
8      X_NEAR = NEAR_NEIGHBORS(x, T, r);
9      foreach x_near in X_NEAR do
10       if Collisionfree(x_near, x_new) && x_near.cost + cost(
             x_near, x_new) < c_min then
11         x_min = x_near;
12         c_min = x_near.cost + cost(x_near, x_new);
13     T.add_egde(x_min, x_new);
14     foreach x_near in X_NEAR do
15       if Collsionfree(x_new, x_near) && x_new.cost + cost(x_new
             , x_near) < x_near.cost then
16         T.del_edge(x_near_parent, x_near);
17         T.add_edge(x_new, x_near);
18     Return Extended;
19   else
20     Return Trapped;
```

Während wie beim Erstellen eines *RRT* auch bei *RRT** zuerst der nächste Nachbar als Vaterknoten festgelegt wird, folgt daraufhin eine Speicherung der nächsten Nachbarn von `x_new` in einem gewissen Radius r in der Liste `X_NEAR`. Es wird `x_min` als der Abstand zum nächsten Knoten gesetzt. Die Funktion `x_nearest.cost` liefert die Kosten, um vom Startknoten zu `x_nearest` zu kommen, zurück, während die Funktion `cost(x_nearest, x_new` die Kosten des Pfades von `x_nearest` zu `x_new` berechnet. Als vorläufiger Startwert beinhaltet `c_min` demnach die Kosten, um vom Startknoten aus zu `x_new` zu kommen.
Die Liste `X_NEAR` wird auf den besten Vaterknoten, also den mit den geringsten Kosten für `x_new`, überprüft. Dieser beste gefundene Nachbar wird für `x_new` als Vaterknoten gesetzt, also eine Kante zwischen `x_new` und `x_near` gezogen.

Nachdem so ein Pfad vom Vaterknoten zu `x_new` gebildet wurde, wird der Baum Neuverknüpft. Bei jedem Knoten innerhalb des Radius von `x_new` wird überprüft, ob die Kosten mit `x_new` als Vaterknoten geringer werden. Wo immer dies der Fall ist, wird `x_new` als Vaterknoten gesetzt.

2.2.2 Vor- und Nachteile RRT*

Der erste Unterschied zu einem *RRT* ist, dass nicht der nächste Nachbar als Vaterknoten gesetzt wird, sondern der mit den besten Kosten. Je nach Wahl des Radius *r* und Güte der Kostenfunktion kann hier einiges an Ümweg"gespart werden.
Der Hauptunterschied ist jedoch die Neuverknüpfung bereits bestehender Knoten über `x_new`. Ein Nachteil des *RRTs* war auch, dass Knoten, die aus bereits gut mit Knoten gefüllten Regionen neu hinzugefügt wurden, den Baum nicht wirklich bereichert hatten. Dies ändert sich nun, denn jeder von Knoten umgebene, neu hinzugefügte Knoten verbessert die Kosten aller Knoten, die durch `x_new` besser erreichbar ist. Dies führt sogar soweit, dass ein *RRT** asymptotisch optimal ist, d.h. bei genügend langer Laufzeit der Pfad zum Optimum konvergiert.Dies ist der Fall, da jeder Knoten entweder hilft, den Baum zu expandieren oder aber für bessere Pfade innerhalb des Baumes sorgt ([vgl. Kapitel 5 in 11]). Je nach Art der Kostenfunktion und dem Samplen neuer Knoten kann der RRT* auch mit bestimmten Eigenschaften ausgestattet werden, z.B. präferiertes Wachsen in bestimmte Richtungen.

2.3 Nicht-holonomische Einschränkungen

Leider ist dieser Algorithmus so nicht eins zu eins auf ein Auto umsetzbar, da dieses bestimmten kinematischen Bedingungen unterworfen ist. Es kann sich zum Beispiel nicht in jede beliebige Richtung bewegen (z.B. seitlich) und hat abhängig vom Lenkradius gewisse Punkte, die nicht in einem Schritt erreichbar sind. Wie die Einschränkungen genau aussehen, welche Rahmenbedingungen an Hardware und Software gegeben waren und inwieweit der RRT*-Algorithmus angepasst werden musste, behandelt das nächste Kapitel.

3 Umsetzung

Bevor wir uns den notwendigen Anpassungen des *RRT**-Algorithmus widmen können, müssen wir die kinematischen und physikalischen Beschränkungen des Autos analysieren. Anschließend wird das verwendete Framework ROS - Robot Operating Systems - vorgestellt, wonach wir uns mit der Wahl einer geeigneten Metrik und Kostenfunktion beschäftigen.

3.1 Hardwareausstattung der Autos

Das Dahlem Center for Machine Learning and Robotics arbeitet mittlerweile mit dem Modelfahrzeug AutoNOMOS Mini v3 (1:10).[TODO Bild anfügen] Der Hauptcomputer auf dem Auto ist ein *Odroid*(XU4 64GB) mit Ubuntu Linux als Betriebssystem und ROS (Robot Operation Systems) als Steuerungssystem [3].

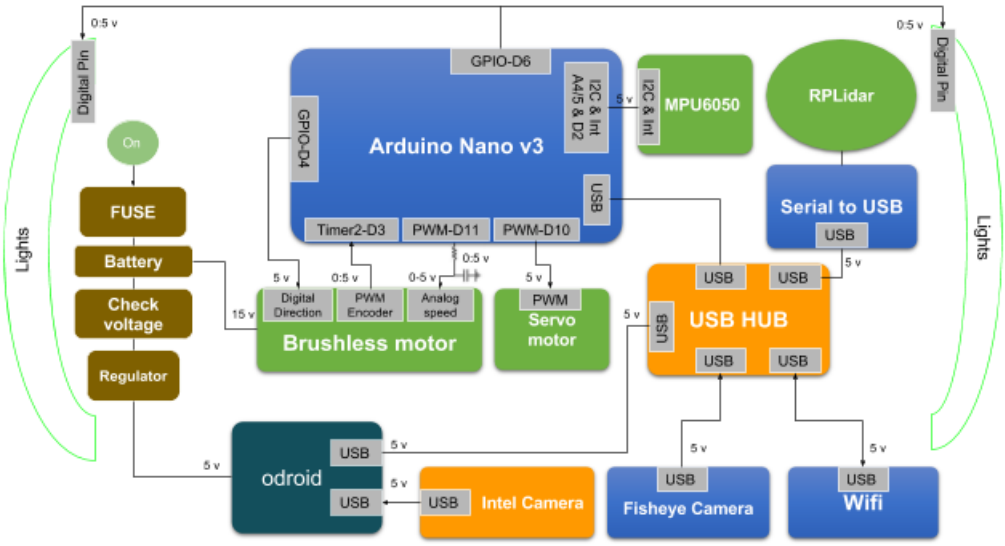

Abbildung 5: Überblick über die Module des AutoNOMOS Mini v3

Motorisiert ist das Auto mit einem bürstenlosen [TODO??] DC-Servomotor FAULHABER 2232. Die Lenkung wird von dem Servomotor HS-645-MG übernommen, beide Motoren werden mithilfe einer *Arduino Nano* Platine gesteuert.
Zur Wahrnehmung der Umgebung besitzt das AutoNOMOS Mini v3 mit dem RPLIDAR A2 360 einen rotierenden Laserscanner, der in der Lage ist, die Umgebung des Autos auf Hindernisse zu überprüfen. Als Rückgabewert liefert der RPLIDAR pro Gradwinkel den Wert, wie weit das nächste Hindernis in dieser Richtung entfernt ist, also insgesamt 360 Werte (einen pro Winkel).
Auf dem oberen Teil des Autos befestigt ist das *Kinect-type stereoscopic system* (Intel RealSense SR300), welches eine Wolke aus 3D Punkten liefert, die dazu benutzt werden kann, Hindernisse zu erkennen. Außerdem kann die Kamera des *Kinect-type* Sensors dazu benutzt werden, Fahrbahnmarkierungen und Objekte direkt vor dem Auto zu lokalisieren.
Der letzte äußere Sensor, auch am oberen Teil des Autos angebracht, ist die Fischaugen-Kamera. Diese zeigt nach oben, zur Decke, und kann dazu benutzt werden bestimmte markante, feststehende Objekte zu lokalisieren, damit das AutoNOMOS Mini v3 sich auch innerhalb von Räumen orientieren kann. Dazu kann eine GPS Navigationseinheit simuliert werden, indem die an der Decke angebrachten vier Lampen in unterschiedlichen Farben leuchten.
Die Sensoren sind entweder via USB 3.0 an der Hauptplatine oder direkt am *Odroid* angeschlossen.
An inneren Sensoren besitzt das AutoNOMOS Mini v3 eine MPU6050, die einen Beschleunigungssensor und ein *Gyroskop* enthält. Mithilfe dieser MPU kann das AutoNOMOS Mini v3 seine Orientierung, seine Richtung im Raum bestimmen. Außerdem können Messungen zur *Odometrie* ergänzt werden.
Das AutoNOMOS Mini v3 wird über eine 14,8 V Batterie mit Energie versorgt.

3.2 Software: ROS - Robot Operating Systems

ROS stellt Bibliotheken und Werkzeuge zur Verfügung, die Software-Entwicklern helfen sollen, Robotik Anwendungen zu kreieren [6]. Unter anderem beinhaltet ROS Gerätetreiber, Bibliotheken, Visualisierungswerkzeuge, Paketmanagement und vieles mehr. ROS ist Open Source und unter der BSD Lizenz verfügbar.

3.2.1 Architektur

Mithilfe von ROS können so genannte *Nodes*, ausführbare Programme, erzeugt werden, die über so genannte *Topics* kommunizieren können. Dies passiert über einen anonymisierten Publisher/Subscriber Mechanismus, das heißt Daten generierende Knoten können auf relevanten *Topics* Nachrichten senden, und interessierte Knoten können von relevanten *Topics* Nachrichten empfangen.
Für jedes *Topic* ist dabei auch die Nachrichtenart definiert, die für dieses *Topic* veröffentlicht und von diesem *Topic* empfangen werden. Dies können neben simplen Datentypen auch komplexe, selbst definierte Datenstrukturen sein. Dabei wird nur dieser eine, vorher festgelegte Datentyp der Nachricht vom *Topic* akzeptiert. *Topics* stellen nur eine unidirektionale Verbindung zur Verfügung. Für die Abwicklung von zum Beispiel Remote Procedure Calls sind sogenannte Services zuständig. Diese ermöglichen, eine Antwort auf eine bestimmte Anfrage nach dem Client Server Prinzip zurückzusenden.

3.3 APIs und Einbettung zu bereits vorhandene Knoten

Das Dahlem Center for Machine Learning and Robotics entwickelte ROS-Pakete für die Steuerung ihrer autonomer Fahrzeuge. Diese Pakete und daraus resultierenden ROS-Nodes können dazu genutzt werden, den von mir entwickelten RRT*-Pfadplaner möglichst gut einzubetten. So kann durch das visuelle indoor GPS die Position des Autos bestimmt werden, die der Pfadplaner für seine Berechnungen braucht. Die resultierende *Trajektorie*, die der Pfadplaner entwickelt, wird einem Steuerungsknoten übergeben, der diese *Trajektorie* in Motorbefehle, also Beschleunigungen und Lenkungen, umsetzt.

3.3.1 Bestimmung der Odometry und visual GPS

Zur Ausführung des Algorithmus gehört, dass das Auto sich selbst lokalisieren kann und seine eigene Startposition feststellen kann. Diese wird, mit der aktuellen Ausrichtung des Autos, dazu benutzt um den Startpunkt für den Algorithmus zu setzen.
Das Dahlem Center for Machine Learning and Robotics hat dafür zwei verschiedene Varianten entwickelt, die kombiniert einander ergänzen können.

1. Odometrie
   Anhand des Lenkwinkels und der Motordrehgeschwindigkeit kann die erwartete Position, Geschwindigkeit und Beschleunigung berechnet werden. Mit dem im Auto verbauten Gyroskop werden Lageänderungen entlang jeder Achse gemessen. Aufgrund mechanischer Beschränkungen und Ungenauigkeiten sind

Messungen mithilfe der Odometrie nie ganz exakt und somit kann eine so bestimmte Position mit der Zeit von der tatsächlichen Postition abweichen.

2. Visual GPS
   Da es innerhalb von Gebäuden Schwierigkeiten mit der genauen Positionsbestimmung via GPS gibt, wurde mithilfe von vier Lampen, die an der Decke angebracht wurden, ein visuelles GPS simuliert. Diese vier Lampen leuchten in unterschiedlichen, gut erkennbaren Farben und werden mit der Fischaugenkamera, die zur Decke ausgerichtet ist, erfasst. Anhand der Lage der Lampen zueinander und wie die Fischaugenkamera diese sieht kann die Position des Autos bestimmt werden.
   Dazu muss jedoch gesagt werden, dass das eindeutige Erkennen der Lampen nur unter guten Bedingungen (Lichtverhältnisse, alle 4 Lampen im Bild) möglich ist.

3.3.2 Low-Level-Planer

Das Dahlem Center for Machine Learning and Robotics entwickelte einen Steuerungsknoten fub controller, der für die Steuerung der Motoren zuständig ist. Dieser Knoten lauscht auf das *Topic* `"planned_path"`. Auf `"planned_path"` kann eine *Trajektorie* publiziert werden, die dann mithilfe des Steuerungsknotens vom Auto abgefahren wird. Dabei kümmert sich dieser Steuerungsknoten allerdings nicht um etwagige Hindernisse, die mit dem Auto kollidieren könnten, sondern fährt nur die Trajektorie ab. Somit muss der Pfadplaner selbst alle Kollisionen mit Hindernissen ausschließen.
Das Format der *Trajektorie* wurde von der Arbeitsgruppe der FU Berlin definiert und besteht aus

- `std_msgs/Header`: Hier wird die aktuelle Zeit gespeichert.
- `string child_frame_id`: [TODO]
- `fub_trajectory_msgs/TrajectoryPoint[] trajectory`: Eine Liste aus Trajektorie-Punkten.

Ein Trajektorien-Punkt symbolisiert einen abzufahrenden Knotenpunkt und wiederum besteht aus

- `geometry_msgs/Pose pose`: Hier sind Position und Orientierung des Autos gespeichert.
- `geometry_msgs/Twist velocity`:Hier wird die Geschwindigkeit des Autos an diesem Punkt gespeichert.
- `geometry_msgs/Twist acceleration` Hier wird die Beschleunigung des Autos an diesem Punkt gespeichert.

Die Position wird in x-Position und y-Position angegeben, ausgehend von einer Ecke des Raumes. Die Orientierung wird durch ein *Quaternion* dargestellt, bei dem durch vier Werte die Drehung in jeder Richtung des Raumes genau bestimmt ist. Allerdings

genügt uns die Drehung um die z-Achse, also die Drehrichtung über die Vertikalachse, weshalb dieses Quaternion in einen Winkel, der sogenannten Gierung(engl. *yaw*), umgerechnet wird.
Nachdem die notwendige Infrastruktur erläutert wurde, folgt eine kurze Betrachtung der kinematischen und physikalischen Einschränkungen, denen das AutoNOMOS Mini v3 unterworfen ist. Anschließend können wir uns endlich dem Kernstück der Arbeit, dem eigentlichen RRT*-Pfadplaner, widmen.

3.4 Kinematische und physikalische Einschränkungen

Im Gegensatz zu holonomischen Fahrzeugen sind die möglichen Pfade eines Autos nicht so einfach zu berechnen. Deshalb stellen wir zuerst zur Orientierung ein mathematisches Modell des Autos auf und definieren die Randbedingungen, unter denen es agiert. Die mathematischen Definitionen der Randbedingungen und des Autos sind in der Bachelorarbeit von Lukas Gödicke [7] gut beschrieben und werden hier (übersetzt) übernommen.

3.4.1 Randbedingungen

Die Position des Autos bezieht sich auf den Mittelpunkt zwischen den Hinterrädern mit den Koordinaten x und y. Der Winkel $\theta \in S, S = [0, 2\pi]$ steht für die Ausrichtung des Autos, der Winkel $\phi \in S$ für die Ausrichtung der Räder (=Lenkwinkel). Wir nehmen an, dass das Auto sich auf einer flachen Ebene bewegt. Somit ist der Arbeitsraum definiert durch $C = \mathbb{R} \times \mathbb{S}$. Ein Zustand des Autos wird beschrieben durch $q \in C = (x, y, \theta)$.
Zur Vereinfachung der Rechnungen wird angenommen, dass die Räder bei Bedarf sich sofort, ohne Zeitverlust, in die gewünschte Position begeben. Da der Lenkwinkel keinen Einfluss auf den Momentanzustand des Autos hat wird dieser in der Zustandsbeschreibung nicht aufgeführt. Zusätzlich kommt noch hinzu, dass das Auto zur Testzwecken bestimmte Geschwindigkeiten nicht überschreiten darf, sodass der minimale Kurvenradius nur durch den Lenkwinkel beschränkt ist und nicht auch noch zusätzlich durch die Geschwindigkeit.

3.4.2 Einschränkungen durch das Auto

Ein Auto hat eine Möglichkeit sich fortzubewegen: Es kann beschleunigen. Eine negative Beschleunigung wäre Bremsen oder sogar Rückwärtsfahren. Zusätzlich zu dieser einen Möglichkeit, den Zustand des Autos zu ändern, kann das Auto das Resultat der Beschleunigung ändern, indem der Lenkwinkel angepasst wird. Dieser reicht von ϕ_{max} bis $-\phi_{max}$, den maximalen Lenkwinkeln des Autos. Je nach Lenkwinkel i_ϕ, Geschwindigkeit i_g und Anfangsausrichtung des Autos θ ändern sich verschiedene Parameter des Zustandes $q(x, y, \theta)$:

$$\dot{x} = i_g cos\phi$$

$$\dot{y} = i_g sin\phi$$
$$\dot{\theta} = \frac{i_g}{L} tan(i_\phi)$$

L ist hierbei der Radstand, also der Abstand zwischen Vorder- und Hinterachse.

3.4.3 Konsequenzen

Ein normaler *RRT** besitzt *Trajektorien* zum Ziel, die für ein Auto aufgrund der oben benannten Beschränkungen nicht abfahrbar sind. So entstehen beispielsweise diskrete Ecken und Kanten die für ein sich kontinuierlich bewegendes Auto unmöglich zu bewältigen ist. Somit ist die erste Aufgabe, die Pfade im *RRT** für das Auto irgendwie abfahrbar zu machen.
Dazu hatte ich am Anfang unterschiedliche Ansätze:

1. RRT* normal ausführen. Sobald ein Pfad zum Ziel gefunden: Diesen so bearbeiten, dass dieser vom Auto zu bewältigen ist

2. RRT* normal ausführen, jedoch mit geeignetem Algorithmus dafür sorgen, dass das Auto jeden Knoten erreichen kann

3. Nur Knoten hinzufügen, die von ihrem Vaterknoten aus unter oben genannten Bedingungen stets erreichbar sind

Das Problem des ersten Ansatzes war, das unter Umständen je nach Umgebung ein Pfad nicht nur wegen bestimmter Ecken und Kanten nicht abfahrbar ist, sondern auch wegen Kurven, die Aufgrund des Lenkradius des Autos nicht zu bewältigen wären. Es wäre zwar möglich, für kurze Zeit die geplante Trajektorie zu verlassen und nach Ende der Kurve wieder zu ihr zu stoßen, allerdings ist es unter diesen Umständen schwierig eine Hindernisfreiheit zu garantieren.
Die zweite Variante war Thema der Bachelorarbeit meines oben bereits erwähnten Kommolitonen David Gödicke [7]. Dieser benutze für den Pfad zwischen zwei Kurven Dubins Curves [5], bei denen drei Einstellungen aus linkem maximalem Lenkeinschlag, rechtem maximalen Lenkeinschlag und geradeaus fahren dreimal hintereinander verkettet werden . Damit kann zwar jeder Punkt des Arbeitsbereiches erreicht werden, allerdings war die Berechnung zu aufwändig und langsam, was für Echtzeit Szenarien im Straßenverkehr ein hartes Ausschlusskriterium ist [vergleiche 7, Kapitel 7].
Deshalb entschloss ich mich zuerst für die dritte Variante, bei der Knoten nur zum RRT*-Graph unter der Bedingung hinzugefügt werden, dass sie erreichbar sind.

3.5 RRT*-Pfadplaner für autonome Fahrzeuge

3.5.1 Vorauswahl

[TODO ist eig alles veraltet, muss neu geschrieben werden. Ich kann mit dem Auto jetzt jeden Punkt außerhalb der Wendekreise erreichen...damit fällt alles mit Erreichbarkeit über den Winkel weg, Projektion, Berechnung der Orientierung und Sicherstellen-das-der-Knoten-nicht-zu-nah-ist bleibt aber bestehen...

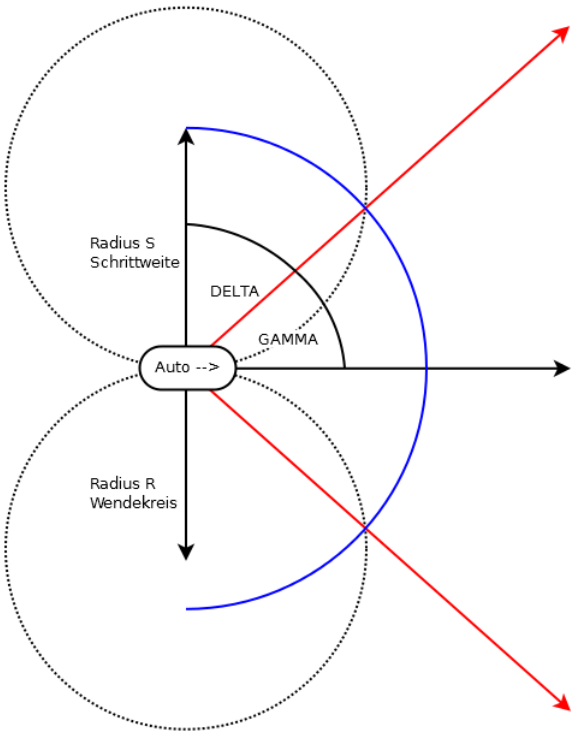

Abbildung 6: Bedeutung der Schrittweite und Radius für den Winkel Gamma

Vor allem müssen neue Grafiken erstellt werden (Mi?)!] Diese beinhaltete, bei der Erstellung des RRT* vom Auto nicht erreichbare Knoten gar nicht erst zuzulassen. Das Auto sollte also von einem Knoten zum nächsten mit nur einer Lenkeinstellung direkt fahren können. Dies sollte die Berechnungszeit stark verringern.
Zum Ausschluss der Knoten verwendete ich zuerst eine Heuristik: Ein nächster Nachbar N für einen neu hinzugefügten Knoten K kam nur dann in Frage, falls dieser den neu hinzugefügten Knoten K auch erreichen konnte. Dazu wurde die Ausrichtung des Autos im potenziellen Elternknoten N mit dem Richtungsvektor zwischen den beiden Knoten verglichen. Somit war K gültig, falls der Winkel zwischen der Ausrichtung von N und dem direktem Weg zum Knoten K (Richtungsvektor) einen bestimmten Wert γ nicht überschritt. γ direkt zu berechnen war nicht ohne weiteres möglich, doch mit Hilfe des Kosinussatzes konnte der Winkel δ (siehe Abbildung 3.5.1) berechnet werden. Dieser berechnet sich aus dem Radius R des Wendekreises des Autos und der verwendeten Schrittweite, also dem maximalen Abstand zwischen zwei Knoten.
Der Knoten K konnte wurde im ersten Schritt als gültig erkannt, wenn für die Differenz zum Winkel des Elternknoten galt:

$$diff(K,N) < \gamma$$
$$\gamma = 90° - \delta$$
$$\delta = arccos(\frac{Schrittweite}{2R})$$

Alle Knoten außerhalb dieses Winkels würden bei der Projektion im Wendekreis des Autos landen, wie in Abbildung 3.5.1 zu sehen. Diese Knoten(rot) wurden verworfen.

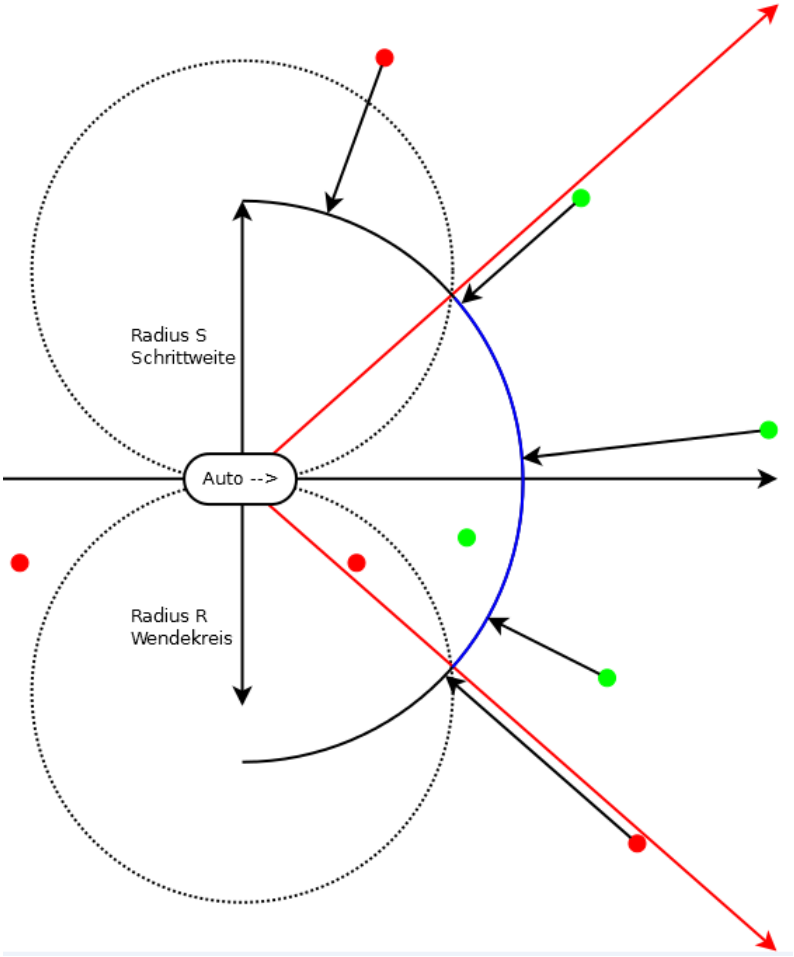

Abbildung 7: Projektion der Punkte auf Schrittweite

Als nächster Schritt wird, wie in Kapitel 2.2 geschildert, der Knoten K an den nächsten (gültigen) Nachbarn N projiziert und der Vaterknoten bestimmt. Sollte der Knoten näher am Vaterknoten dran sein, als die Schrittweite ist, wird der Knoten nicht projiziert, da sonst alle Knoten zu ihrem Vaterknoten den gleichen Abstand hätten und bestimmte Bereiche somit nicht auf optimalen Weg erreichbar wären. Da innerhalb der Schrittweite durch den maximalen Lenkwinkel des Autos selbst die Knoten, die vom Winkel her gültig sind, nicht erreichbar sein können ist eine zusätzliche Überprüfung für diese Knoten notwendig.
Da der maximale Lenkwinkel eines Autos vorgegeben ist hängt also die Erreichbarkeit des Knotens allein von der Schrittweite ab. Mit dieser kann auch vorgegeben werden, wie um wie viel Grad sich die Ausrichtung des Autos bei einem Schritt ändern darf und ob beispielsweise Kehren erlaubt sind. Dabei muss jedoch beachtet werden: Bei einer zu kleinen Schrittweite werden viele Knoten verworfen und es werden kaum enge Kurven möglich sein. Bei einer zu großen Schrittweite werden sehr kurvige Trajektorien entstehen, sodass z.B. Hinderniserkennung nicht mehr so einfach ist.

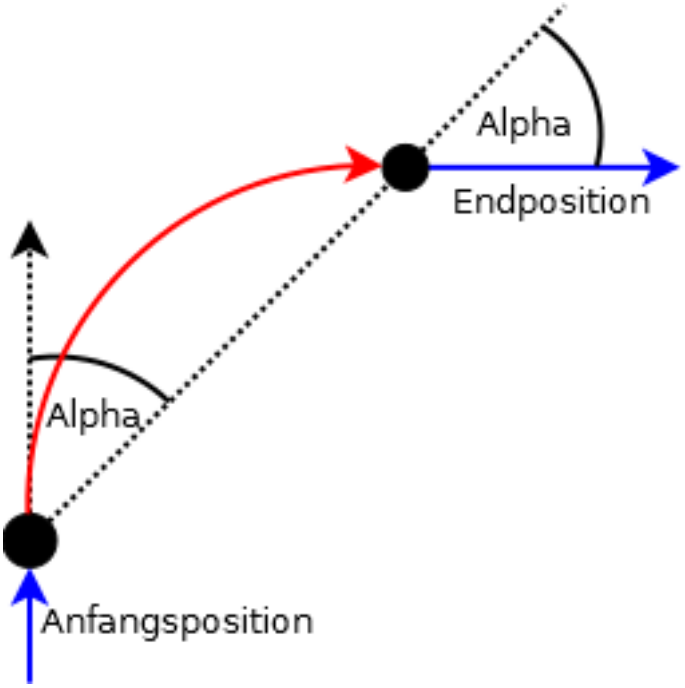

Abbildung 8: Berechnung der Ausrichtung

3.5.2 Bestimmung des Vaterknotens

Nachdem der Knoten an seinen nächsten Nachbarn projiziert wurde, wird nun aus allen nächsten Nachbar innerhalb eines Radius R der mit den besten Kosten gewählt. Im Verfahren zur Bestimmung des nächsten Nachbarn, siehe 3.5.1, wurden die Punkte innerhalb der Schrittweite nicht berücksichtigt. Es ist dem Auto nicht möglich, vom Vaterknoten N zum Knoten K zu fahren, wenn K im Wendekreis des Autos liegt.
Für jeden Knoten im Radius R wird geprüft, ob dieser Knoten unseren neuen Knoten K erreichen kann. Das wird ausgerechnet, indem vom potentiellen Vaterknoten N aus zwei Mittelpunkte der Wendekreise bestimmt werden. Wenn der Abstand von K zu diesen Mittelpunkten kleiner ist als der Radius der Wendekreise, liegt K im Wendekreis und ist nicht erreichbar. Damit kommt N als Nachbar nicht in Frage. Falls kein nächster Nachbar im Radius R den Knoten K erreichen kann, wird K verworfen.
Nachdem erfolgreich ein Vaterknoten bestimmt und dem neu hinzugefügtem Knoten K zugewiesen wurde, muss noch die Ausrichtung oder Orientierung bestimmt werden, die das Auto im Knoten K hat.

3.5.3 Bestimmung der Orientierung

Da das Auto im Vaterknoten N eine bestimmte Ausrichtung hat und mit nur einer Lenkeinstellung zum nächsten Knoten K gelangt, ist dort die Ausrichtung dadurch festgelegt und berechnet sich aus der Ausrichtung des Vaterknotens, Winkel θ, und dem Richtungsvektor von N nach K. Der Richtungsvektor kann in einen Winkel relativ zu x-Achse mit Hilfe des Skalarprodukts ausgerechnet werden. Wie in der Zeichnung 3.5.3 zu sehen ist, wird der Winkel des Richtungsvektors zweimal auf die Ausrichtung

im Vaterknoten addiert, um die Ausrichtung im neuen Knoten zu erhalten.

3.6 Metrik und Kostenfunktion

Eine besondere Bedeutung zur Erzeugung einer "guten"Trajektorie kommt der Kostenfunktion zu. Diese sorgt nicht nur dafür, welcher Knoten als Vaterknoten ausgewählt wird, sondern spielt auch beim Neuverknüpfen des Baumes eine Rolle.Somit kann mithilfe der Kostenfunktion für besonders gut abfahrbare oder besonders kurze Pfade gesorgt werden.
Da durch die Orientierung des Vaterknotens die Orientierung des Kindknotens bereits festgelegt ist (siehe Abbildung 3.5.3, können aus eigentlich geraden Strecken sehr ungünstige Pfade entstehen.
In Abbildung 3.5.1 wird als Kostenfunktion der euklidische Abstand benutzt. Knoten C wählt aus B1 und B2 den nächsten Nachbarn aus, das ist B2. Leider entsteht dadurch eine sehr kurvige Route, bei der vom fast maximalem Lenkeinschlag nach rechts auf den maximalen Lenkeinschlag der anderen Seite gewechselt werden muss. Neben Nachteilen des Komforts, der Sicherheit und längerem Weg wird auch die Ungenauigkeit höher. Anfangs wurde angenommen, dass der Lenkwinkel sich sofort ändern kann. Während bei kleinen Änderungen diese Annahme nur für kleine Fehler sorgt, sind die Auswirkungen bei solch starken Änderungen deutlicher spürbar.

In Abbildung 3.6 hingegen wurde eine bessere Kostenfunktion gewählt, sodass über B1 gehend ein effizienterer, kürzerer Pfad entsteht. Die Kostenfunktion muss dabei die Ausrichtung des Autos beim Vaterknoten N berücksichtigen. [TODO Auswahl] Eine Möglichkeit wäre dann, die tatsächliche zu fahrende Strecke des Autos zu benutzen und dadurch effizientere Pfade zu bevorzugen. Eine andere Möglichkeit ist, zu harte Richtungsänderungen zu bestrafen und kleine $\Delta\theta$ zu bevorzugen.
Als Lösung wurde hierbei eine Kombination aus dem euklidischem Abstand und der Winkeldifferenz der Knoten K und N zur Kostenberechnung zu benutzen. Die Kosten berechnen sich dann wie folgt:

$$cost(K) = cost(N) + eukl.Abstand(K,N) * (1 + Winkeldifferenz)$$

Bei einer geraden Strecke wird lediglich der euklidische Abstand als Kosten aufsummiert, bei einer Kehre von 180° bis zum vierfachen der euklidischen Kosten (Winkeldifferenz π +1).
Nachteil dieser Kostenfunktion ist, dass nicht berücksichtigt wird das starke Kurven weniger ßchlimmßind auf größerer Strecke. Mit anderen Worten, das Fahren genau auf den minimalen Wendekreisen ist meist weniger angenehm als das Fahren auf weiten Kreisen, auch wenn beide Pfade die gleiche Winkeldifferenz haben. Das Problem hierbei ist, dass ein kleinerer Kreis geringere euklidische Kosten hat und demzufolge gegenüber dem größeren Kreis bevorzugt wird.

3.6.1 Rewiring

[TODO Rewiring neu schreiben, siehe oben. Hat sich geändert, Rewiring muss nicht Rekursiv durchgeführt werden.] Der Knoten K wurde neu in den Baum hinzugefügt.

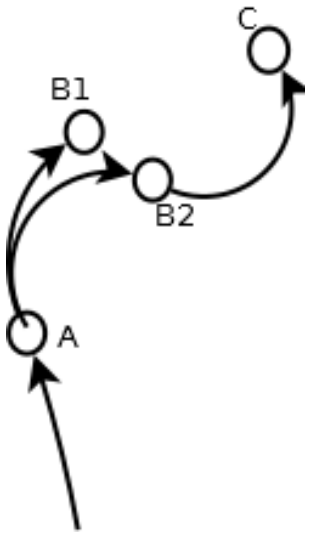

Abbildung 9: einfache euklidischeKostenfunktion

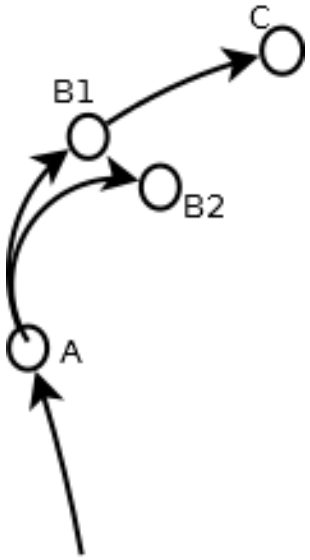

Abbildung 10: erweiterte Kostenfunktion

Nun wird überprüft, ob bereits vorhandene Knoten besser erreichbar sind, wenn der Weg über K gewählt wird. Dazu werden zunächst im Radius R alle in Frage kommenden Nachbarn ausgewählt. Aussortiert werden alle, die entweder vom Knoten K aus nicht erreichbar sind (gleiche Überprüfung wie oben) oder aber bei denen K keine Verbesserung bewirkt also die Kosten über den Knoten K gleich oder kleiner sind.
Wenn man von den jetzt noch ausgewählten Knoten M einfach K als Vater hinzufügen würde, müsste jeweils die Orientierung der Knoten, der Winkel θ, geändert werden. Das Auto gelangt auf anderem Wege zu diesen Knoten und hat somit eine andere Ausrichtung als vorher. Allerdings kann es dadurch vorkommen, das wenn der Knoten $m \in M$ eine neue Orientierung hat, Kinder von m nicht mehr durch m erreichbar sind [TODO Zeichnung]. Deshalb wird, anstatt die Orientierung zu ändern, einfach ein neuer Knoten hinzugefügt, der zwar die gleicher Koordinaten hat wie m, aber eine andere Orientierung. Dies wird mit allen von K erreichbaren Knoten

durchgeführt. Alle somit zum Baum neu hinzugefügten Knoten sind durch K besser erreichbar und können nun selbst zur Neuverknüpfung des Baumes beitragen. Somit wird der Algorithmus des Neuverknüpfens rekursiv auf alle im vorherigen Schritt durch das Rewiring neu hinzugefügten Knotens angewendet.
Der Vorteil dieses Mechanismus ist, dass mögliche Optimierungen schnell durch den Baum wandern und keine Knoten ungültig werden. Außerdem müssen nicht so viele randomisierte Punkte erzeut werden, was der Laufzeit zu Gute kommt. Nachteile sind allerdings eine geringere Anzahl von räumlich verteilten Punkten (es liegen Punkte "übereinander"). Zudem kann dieses Verfahren viel Zeit beanspruchen, ohne viel Effekt zu haben, wenn besonders viele Punkte im Baum existieren und Bereiche weit abseits der Zielregion neu verknüpft werden.

3.7 Datenstruktur

RRT* hat eine amortisierte Laufzeit von O(n log(n)). Um diese Laufzeit auch in der Praxis zu erreichen, muss auf spezielle Datenstrukturen zurückgegriffen werden. Insbesondere müssen mehrfach sowohl der nächste Knoten als auch die nächsten innerhalb eines Radius ermittelt werden. Für die sogenannte *k-nearest neighbour* Suche und *radius k-nearest neighbour* existieren bereits viele wissenschaftliche Untersuchungen und auch empfohlene Datenstrukturen, wie zum Beispiel k-d-Bäume [2]. Leider benötigt der RRT* Algorithmus eine dynamische Datenstruktur, da Punkte erst nach und nach hinzugefügt werden. Eine weitere Einschränkung waren fehlende Implementierungen, die auf die Anwendungsfälle von RRT* zugeschnitten waren.
TODO Gitter wird nicht verwendet, sondern einfache Liste. Vllt Kombination aus beidem? Aufgrund dieser Einschränkungen wurde keine komplizierte, unter Umständen bessere Datenstruktur gewählt. Stattdessen wurde ein einfaches Gitter implementiert, deren Achsen die x- und y-Werte des einzufügenden Punktes repräsentierte. Dadurch fallen Punkte, die nah beeinander sind, in die gleiche oder benachbarte Zellen. Je nach Wahl der Größe der Zellen müssen nur die Knoten der Zelle selbst und der Nachbarzellen überprüft werden. Ein weiterer Vorteil des Gitters ist das schnelle Einfügen eines Knotens: Dadurch dass das Gitter statisch bleibt können Arrays als Rahmen gewählt werden und die Laufzeit zum Einfügen von Knoten liegt bei O(1).
Solange allerdings noch sehr wenige Knoten sich im Gitter befinden, muss unter Umständen das gesamte Gitter nach Nachbarn abgesucht werden, was die Laufzeit stark verschlechtert. Innerhalb einer Zelle sind die Knoten in einer Liste gespeichert. Sollten sehr viele Knoten in eine Zelle fallen, dauert das Durchsuchen dieser Zelle sehr lange.

3.8 Dokumentation der Durchführung und entstandener Artefakte

[TODO] Anfangs wurde Python code programmiert. Dieser war weder objektorientiert noch besonders schnell, obwohl mit numpy eine für Vektorberechnungen optimierte Bibliothek benutzt wurde. Dann wurde - aufgrund von Performancegründen - zur Programmiersprache C++ gewechselt. Dabei wurde auch Objektorientierung eingeführt, um die einzelnen Knoten in ihrer Komplexität besser handhaben zu können. Dies verschlang sehr viel Zeit.
Als Datenstruktur wurde, sowohl eine Liste als auch ein Gitter verwendet. Zur Ge-

schwindigkeitsoptimierungen wurde die Header-Bibliothek Eigen 3 benutzt. [TODO mehr schreiben wenn was funktioniert]

3.9 Beschreibung besonderer Schwierigkeiten und wie diese umgangen wurden

[TODO] evtl in nächstes/übernächstes kapitel. Lernfortschritte deutlich machen/ was habe ich aus den jeweiligen Problemen gelernt? Alles noch ergänzen...

3.9.1 Schwierigkeiten beim Einbinden bereits bestehender Architekturen

- Einbindung Eigen Bibliothek (ging von der Schwierigkeit her)
- Eclipse (hat viele Nerven gekostet, 4-6 Tage insgesamt zum Funktionieren mit c++11 und allen ROS-Abhängigkeiten
Erkenntnis, das eine mächtige Entwicklungsumgebung auch ein Stein am Fuß sein, mehr eine Behinderung als Hilfe wenn nicht richtig eingebunden, aber auch verweis auf zahlreiche vorteile (debuggen, schnell zwischen fkt hin und her springen)
- visual Gps: Lampen wurden nicht richtig erkannt, viele Hardgecodete Konstanten, funktioniert nicht -> 3 Tage verschwendet
- fub controller: Hart, herauszufinden was in welchem Format dieser controller braucht (z.B. Timestamp), kaum Doku. Und z.T. falsche Konstanten? (Umrechnung der Geschwindigkeit von m/s zur Motorgecshwindigkeit)

3.9.2 Schwierigkeiten beim Programmieren und Kreieren eigener Strukturen

- winkel und flüchtigkeitsfehler
- Programmieren in einer unbekannten programmiersprache (c++), Zeiteinschätzungen

3.9.3 Schwierigkeiten bei Tests und Übertragung auf das Auto

Hehe das kommt noch

3.9.4 Tests und Testdatensätze/Szenarien für die Software)

Test auf ....[TODO] wenns läuft...

3.9.5 Korrektheitsbeweise

3.10 (Evaluation - nur wenn ich dafür Zeit habe)

vllt rechtfertigung warum ich zu faul war/ nix geschafft habe??

4 Zusammenfassung

[TODO]
Hier werden dann Ergebnisse bekannt gegeben, Bilder, Grafiken angefügt die den Erfolg/Misserfolg des Alg. RRT* deutlich machen. ...

5 Ausblick und Fazit

Hier kommt die Zusammenfassung aller Ergebnisse hin, Fazit, Bewertung der Arbeitsweise und der Ergebnisse, was habe ich mitgenommen, was kann ich zurückgeben

5.1 Ausblick

Weitere Forschungen, mit Entfernungsnorm rumspielen, z.B. Manhattan Norm anstatt Euklidische Norm: Bessere Laufzeit, Belohnt geradeausfahren und 90 Grad Turns
RRT*-Smart: Sampling zum ziel hin, bei Problemregionen (nahe an Hindernissen)
Bessere, optimierte dynamische Datenstruktur für die Knoten (Vorschläge machen, kd-tree, quadtree, ...)

Literaturverzeichnis

[1] Nancy M. Amato and Yan Wu. A randomized Roadmap Method for Path and Manipulation Planning. In *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, ICRA, pages 113–120. IEEE, 1996.

[2] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[3] Arbeitsgruppe Robotics FU Berlin. Hardware (AutoNOMOS Model v3). https://github.com/AutoModelCar/AutoModelCarWiki/wiki/Hardware-(AutoNOMOS-Model-v3).

[4] C.Rösmann, F.Hoffmann, and T.Bertram. Timed-elastic-bands for time-optimal point-to-point nonlinear model predictive control. In *European Control Conference (ECC)*, 2015.

[5] Lester Eli Dubin. ON PLANE CURVES WITH CURVATURE, 1961.

[6] Open Source Robotics Foundation. Robot Operating System. http://wiki.ros.org/.

[7] Lukas Gödicke. Rrt based path planning in static and dynamic environment for model cars, March 2018.

[8] Jean-Claude Latombe Jerome Barraquand. Robot Motion Planning: A Distributed Representation Approach, 1991.

[9] Steven M. LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[10] Steven M. LaValle, James J. Kuffner, and Jr. Rapidly-exploring random trees: Progress and prospects, 2000.

[11] S.Karaman and E.Frazzoli. Incremental sampling-based algorithms for optimal motion planning. In *Robotics: Science and Systems*, 2010.

A Anhang

Quellcode der LaTeX-Klasse `node.cpp`:[1]

```
/*
 * Node.cpp
 *
 *  Created on: 19.02.2018
 *      Author: Arbeit
  * @Charlie ab hier musst du nicht mehr weiterlesen,
 * es sei denn dich interessiert wie das Programm funktioniert ;-)
 */

#include "Node.h"
#include <iostream>
#include "const.h"
#include <math.h>
#include <cmath>
#include <iomanip>

using namespace Eigen;


auto Node::calculate_yaw_and_cost()->void {
  //calculate AND SETS yaw and costs
  //TODO check berechnung wenn ausgeschlafen
  Vector2d dir_par;
  Vector2d dir_new;
  //TODO write test has to be null!!
  double par_yaw = parent->get_yaw();
  dir_new = get_dir_vector().normalized();
  parent->calculate_direction(dir_par);
  double alpha = acos(dir_par.dot(dir_new));
  yaw = fmod((2 * alpha + par_yaw), (2 * M_PI));
  double eukl_cost = dir_vector.norm();
  //mehrkosten: Eukl bei gerader Strecke, 7mal euklid bei 360 grad
  double real_cost = parent->get_cost()
      + eukl_cost * (1 + (2 * M_PI - alpha));
  cost = real_cost;
}
auto Node::calculate_cost(const Node&parent) const->double {
  //calculate the costs from parent to this without changing anything
  Vector2d dir_par;
  Vector2d dir_new;
  calculate_direction(dir_new);
  parent.calculate_direction(dir_par);
  double alpha = acos(dir_par.dot(dir_new));
  double eukl_cost = dir_vector.norm();
  //mehrkosten: Eukl bei gerader Strecke, 7mal euklid bei 360 grad
  double real_cost = parent.get_cost()
      + eukl_cost * (1 + (2 * M_PI - alpha));
  return real_cost;
}

```

[1] Es ist nicht üblich, den gesamten produzierten Quellcode bei einer Abschlussarbeit in Textform abzugeben. Werde das auch noch kürzen und nur die wichtigsten FUnktionen reinpacken

```
auto Node::calculate_direction(Vector2d&direct) const->void {
  //Einheitskreis: normalisiert
  direct[0] = cos(yaw);
  direct[1] = sin(yaw);
}
auto Node::calculate_yaw_from_vec(Vector2d& vec) const->double {
  //deprecated

  //precondition: vec has to be normalized
  Vector2d dir = vec;
  double ret_yaw;
  //y is positive, yaw from [0-M_PI)
  if (dir[1] > 0) {
    ret_yaw = std::acos(dir[0]);
  } else {
    //y is negative, yaw from [M_PI-2*M_PI
    ret_yaw = 2 * M_PI - std::acos(dir[0]);
  }

  return ret_yaw;
}
auto Node::calculate_dir_vector(Node& node, Vector2d& dir) ->bool {
  //Direction Vector from Node to *this
  dir = this->get_coordinates_fast() - node.get_coordinates_fast();
  /*  if (dir.isZero()) {
    std::cout << "calculate dir vector: Warning: dir_vector is null! "
        << std::endl;
    return false;
   }*/
  return true;
}
auto Node::calculate_cost()->bool {
  //deprecated
  //Precondition: Node is complete, e.g. correct yaw, coor, parent etc
      .
  double eukl_cost = dir_vector.norm();
  double yaw_diff = std::abs(double(parent->get_yaw() - yaw));
  //yaw flips from 2Pi to 0, diff from PI/4 to 7/4 *PI is PI/2
  if (yaw_diff > M_PI) {
    yaw_diff = 2 * M_PI - yaw_diff;
  }
  cost = parent->get_cost() + eukl_cost * (1 + yaw_diff);
  if (cost != 0) {
    return true;
  } else {
    return false;
  }
}

auto Node::reached_goal()->bool {
  if (!(coordinates[0] < GOAL_AREA[0]) && !(coordinates[1] < GOAL_AREA
      [1])
      && !(coordinates[0] > (GOAL_AREA[0] + GOAL_AREA[2]))
      && !(coordinates[1] > (GOAL_AREA[1] + GOAL_AREA[3]))) {
    return true;
  }
  return false;
```

```
}
auto Node::is_reachable(const Node& parent, const Vector2d& dir_v)
    const->bool {
  //deprecated
  //Tests if parent can reach *this (Node ist reachable from parent)
  Vector2d dir = dir_v.normalized();
  double par_yaw=parent.get_yaw();
  /*  if (std::abs((double) (dir.norm() - 1)) > ACCURACY) {
    std::cout << "Warning! Direction Vector is not normalised! \n"
        << dir
        << std::endl;
   }*/
  double dir_yaw = calculate_yaw_from_vec(dir);
  double diff = abs(dir_yaw - par_yaw);
  if (diff > M_PI) {
    diff = 2 * M_PI - diff;
  }
  if (diff > MAX_ANGLE){
    return false;
  } else {
    return true;
  }

}
auto Node::not_to_near(Node& parent) const->bool {
  /*
   * DOC
   * Funktion checks if node *this
   * is to near for his parent
   * TODO check correctness with examples
   */
  Vector2d left_circle_center, right_circle_center;
  //calculate orthogonal vectors to yaw-anglefloat
  double par_yaw = parent.get_yaw();
  Vector2d par_coor = parent.get_coordinates_fast();
  double x = -sin(par_yaw);
  double y = cos(par_yaw);
  left_circle_center << x, y;
  left_circle_center *= STEERING_ANGLE_RADIUS;
  right_circle_center = left_circle_center * (-1);
  left_circle_center += par_coor;
  right_circle_center += par_coor;
  double distance_right = (coordinates - right_circle_center).norm();
  double distance_left = (coordinates - left_circle_center).norm();
  if (distance_right < STEERING_ANGLE_RADIUS
      || distance_left < STEERING_ANGLE_RADIUS) {
    return false;
  }
  return true;
}
auto Node::check()->bool {
  if (coordinates.isZero()) {
    std::cout << "Check: Coordinates are Zero" << std::endl;
    return false;
  }
  http: //www.cplusplus.com/reference/memory/shared_ptr/
  if (yaw < 0 || yaw > 2 * M_PI) {
```

```
    std::cout << "Check: Yaw is negativ or to big: " << yaw << std::
        endl;
    return false;
  }
  if (parent == nullptr) {
    std::cout << "Check: Parent is nullptr!" << std::endl;
    return false;
  }
  if (validation == Val::invalid || validation == Val::unknown) {
    std::cout << "Check: Validation is unkown or invalid" << std::endl
        ;
    return false;
  }
  if (cost <= 0) {
    std::cout << "Check: Costs are 0 or negative" << std::endl;
    return false;
  }
  if (dir_vector.norm() < 0 || dir_vector.norm() > 2 * RANGE
      || dir_vector.norm() == 0) {
    std::cout << "Check: Dir_vector ist seltsam: " << dir_vector
        << std::endl;

  }
  return true;
}

auto Node::project_to_parent(Node&parent)->bool {
  //Effects: new coordinates and new dir_vector
//  std::cout << "Entering proeject to parent..." << std::endl;
  /*
  if (&parent == nullptr) {
    std::cout << "project to parent: Parent is null!" << std::endl;
    return false;
  }
  if (dir_vector.norm() < STEPSIZE) {
    //Node ist bereits nah genug dran
    std::cout << "project to parent: Near enough" << std::endl;
    return true;
   }*/
  //it is ok to change dir_vector, because node has new coordinates
  //projected if to far away or to close, innerhalb der wendekreise
  if (dir_vector.norm() > STEPSIZE || !not_to_near(parent)) {
    dir_vector.normalize();
    dir_vector *= STEPSIZE;
    coordinates = parent.get_coordinates_fast() + dir_vector;
  }

  return true;
}

auto Node::get_validation() const->Val {
  return validation;
}
auto Node::get_yaw() const->double {
  //TODO Fehler auf Node spezifizieren
  /*  if (yaw<0) {
    std::cout << "Node::get_yaw: Warning! yaw is negative! " << std::
```

```
        endl;
    }*/
  return yaw;
}
auto Node::get_coordinates_fast() -> Vector2d& {
  return this->coordinates;
}
auto Node::get_parent_pointer() const->Node* {
  /*if (parent == nullptr) {
    std::cout << "Node::get_parent_pointer: Warning! Node has no
        parent! "
        << std::endl;
    }*/
  return parent;
}
auto Node::get_cost() const->double {
  /*if (cost == 0) {
    std::cout << "Node::Get_costs: Warning! Node has no costs!"
        << std::endl;
    }*/
  return cost;
}
auto Node::get_dir_vector() const->Vector2d {
  return dir_vector;
}
auto Node::get_coordinates() const->Vector2d {
  return coordinates;
}

auto Node::set_parent(Node& node)->void {
  if (&node == nullptr) {
    std::cout << "Node::set_parent: Warning! parent is set to null"
        << std::endl;
  }
  this->parent = &node;
}
auto Node::set_dir_vector()->bool {
  //Preconditions: Node has coordinates, node has parent
  dir_vector = this->coordinates - parent->get_coordinates_fast();
  if (dir_vector.isZero()) {
    std::cout << "Node::set_dir_vector(): Warning! dir_vector is Zero"
        << std::endl;
    return false;
  } else if (dir_vector.norm() <= 0) {
    std::cout << "Node::set_dir_vector(): Norm is terrible wrong!"
        << std::endl;
    return false;
  }
  return true;
}
auto Node::set_dir_vector(Vector2d& dir)->void {
  if (dir.isZero()) {
    std::cout << "Node::set_dir_vector(): Warning! dir_vector would be
         Zero"<<std::endl;
    return;
  }
  dir_vector=dir;
```

```
}
auto Node::set_coordinates(Vector2d& coor)->void {
  coordinates = coor;
}
auto Node::set_validation(Val enumVal)->void {
  validation = enumVal;
}
auto Node::print_node() const->void {
  std::cout << "_________________________" << std::endl;
  std::cout << "Coordinates: " << coordinates << std::endl;
  std::cout << "Yaw: " << yaw<< std::endl;
  std::cout << "Parent: " << parent << std::endl;
  std::cout << "Costs to node: " << cost << std::endl;
  std::cout << "Direction vector: " << dir_vector << std::endl;
  std::cout << "-------------------------" << std::endl;

}
auto Node::print_node_short()->void {
  std::cout << "Node with cost: " << cost << std::endl;
}
auto Node::equals(Node&b) ->bool {
  if (this->get_coordinates_fast() == b.get_coordinates_fast()
      && this->get_yaw() == b.get_yaw()) {
    return true;
  }
  return false;
}
Node::Node(const Node&node) {
  this->coordinates = node.get_coordinates();
  this->cost = node.get_cost();
  this->dir_vector = node.get_dir_vector();
  this->parent = node.get_parent_pointer();
  this->validation = node.get_validation();
  this->yaw = node.get_yaw();
}
Node::Node(Vector2d& coor) :
    coordinates(coor) {
  if (coor.isZero()) {
    std::cout << "Warning: Node is initialized with 0" << std::endl;
  }
  yaw = 0;
  cost = 2 * RANGE;
  parent = nullptr;
  validation = Val::unknown;
  dir_vector << 0, 0;
}
Node::Node() {
  //default values
  coordinates << 2 * RANGE, 2 * RANGE;
  yaw = 0;
  cost = 2 * RANGE;
  parent = nullptr;
  validation = Val::unknown;
  dir_vector << 0, 0;
}
Node::Node(Vector2d coor, double alpha, Node* parent, Vector2d dir,
    Val val,
```

```
    double c) :
    coordinates(coor), yaw(alpha), parent(nullptr), validation(val),
        cost(
        c), dir_vector(dir) {
}

Node::~Node() {
}
```