Pure Pursuit und RRT

J. Greten, J. Bahn, S. Fleischmann, K. Poeschel

VORWORT

Mittlerweile ist die Forschung des Autonomen Fahrens von der bloßen Vorstellung in die Wirklichkeit gerückt und so weit fortgeschritten, dass dieser Traum schon bald Alltag sein könnte. Bevor dies aber sein kann müssen jedoch etliche Herausforderungen bewältigt werden. Eine davon ist, das Auto sicher und effizient von einem Ort zum anderen zu bringen. Um diese Aufgabe zu meistern, existieren unterschiedliche Ansätze, um dem Auto je nach Anwendungsfall bei gegebenem Ziel eine Trajektorie vorzuschlagen. Die jeweiligen Algorithmen unterscheiden sich in Ausführungszeit, Genauigkeit, Sicherheit und berechnen unterschiedlich optimale Pfade. Im Folgenden haben wir uns für einen Rapidly-exploring random tree (RRT) und seinem Upgrade RRT* entschieden um eine Trajektorie zu erzeugen. Um diese Trajektorie abzufahren wird eine Regelung benötigt, welche Lenkwinkel und Geschwindigkeiten kontrolliert. Hierfür nutzen wir einen Pure Pursuit Controller (PP). Die Karte inklusive Hindernisse wird mit Simultaneous Localization and Mapping (SLAM) generiert und die Position wird mithilfe des im Robotiklabors vorhandenen Kamerasystem VICON bestimmt.

I. VERWENDETE SOFTWARE

Unser Projekt haben wir in ROS mit C++ entwickelt. Sowohl RRT als auch Pure Pursuit sind Algorithmen, die relativ einfach zu verstehen sind, da man sie sich gut bildlich vorstellen kann. Aus diesem Grund ist es aber auch fast notwendig, zum Testen und Debuggen des Algorithmus eine vernünftige Visualisierung zu haben. Bei RRT sollte man sich die erstellte Baumstruktur und bei Pure Pursuit ist der berechnete Wendekreis, sowie dessen Schnittpunkte mit dem berechneten Pfad anzeigen lassen.

Dies hat sich für uns als eine unerwartete Herausforderung herausgestellt, da wir zuvor wenig Erfahrung mit C++ hatten und die Verwendung von ROS das Einbinden von Frameworks wie Qt verkompliziert. Unsere Ursprüngliche Idee war es, das Projekt mithilfe von Gazebo zu simulieren und die Algorithmen ebenfalls in Gazebo zu visualisieren. Da Marker aber in Gazebo nur sehr umständlich zu benutzen sind und wir keinen guten Weg gefunden haben neue Marker während der Programmlaufzeit dynamisch hinzuzufügen, haben wir uns entschieden Gazebo nicht zu benutzen. Anschließend haben wir probiert eine eigene simple graphische Oberfläche für die Visualisierung in Qt zu entwickeln. Hierbei haben wir aber wie schon erwähnt Schwierigkeiten mit der Kombination aus ROS und Qt gehabt.

Stattdessen haben wir jetzt unsere Visualisierung und Simulation mithilfe von rviz erstellt. Dynamische Marker sind hier wesentlich leichter zu implementieren als in Gazebo. Wir haben einen Skeleton-Code der University of Pennsylvania als Grundlage benutzt, da hier im 2019-Skeleton ebenfalls rviz für die Simulation benutzt wird. Der Skeleton-Code ist dafür vorgesehen, dass Studenten unter anderem einen PP-Controller und RRT implementieren. Die Aufgabenstellung für die Studenten war hier allerdings anders als unser Projektziel da die Studenten hier eine rein reaktive RRT-Pfadfindung anhand von den Lidar-Messungen implementieren sollten. Den 2019-Skeleton-Code haben wir also in erster Linie für die rviz-Simulationsumgebung benutzt. Da das 2019-Skeleton keine Möglichkeit bietet, das Programm auf dem Auto selbst auszuführen, statt es zu simulieren, haben wir die ROS-Nodes für das Lidar, Motorkontrolle, Joystick-Driver usw. aus dem 2018-Skeleton benutzt. Das ist auch der gleiche Skeleton-Code den wir im Praktikum Äutonomes Fahren"benutzt haben. Außerdem haben wir für die Visualisierung der RRT-Baumstruktur eine Klasse aus dem F110-RRT-Skeleton-Code von der University of Pennsylvania benutzt.

II. DIE KARTE

Um die Karte zu generieren haben wir Hector SLAM benutzt. Dafür hatten wir entsprechende Config-Dateien von der University of Illinois at Urbana-Champaign gefunden, die bei uns sehr gut funktioniert haben. Wir hatten auch Cartographer ausprobiert, aber dort hatten wir Probleme mit der Lokalisierung.

Ein Problem beim Ausführen von RRT und RRT* war, dass der berechnete Pfad sehr nah an den Hindernissen auf der Karte liegt. In diesen Fall würde das Auto mit der Kante Kollidieren, da das Auto zu breit ist und der Pure Pursuit Algorithmus Kurven innen schneidet und somit durch das Hindernis (Kante) fahren würde. Ein weiteres Problem, welches beim Ausführen des Algorithmus aufgetreten ist, ist das lückenhafte Karten durch das Generieren durch SLAM entstehen. Diese Lücken werden von RRT bzw. RRT* als möglichen Weg wahrgenommen. Auch richtig erkannte Lücken zwischen zwei Hindernissen sind problematisch, da RRT nicht unterscheidet ob die Lücke breit genug für das Auto ist.

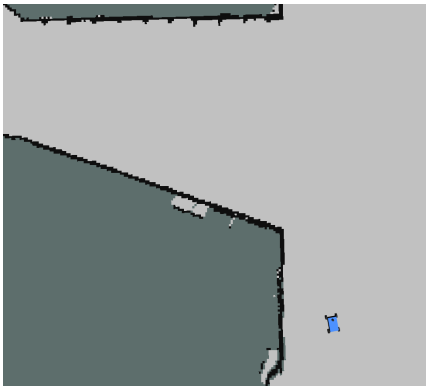

Abbildung 1. nicht gepufferte Karte

Als Lösung für diesen Problem haben wir eine buffed Map erzeugt indem wir alle uns bekannte Hindernisse (Abb. 1 schwarz) auf der Karte vergrößert haben (Abb. 2 grün) und somit rechnen RRT und RRT* mit einem größeren Hindernis als eigentlich vorliegt. Der von uns gewählte Vergrößerungsfaktor beruht auf dem trial and error Prinzip und muss kartenabhängig neu eingestellt werden. In Rviz wird sowohl die normale Karte als auch die vergrößerte Karte übereinander angezeigt.

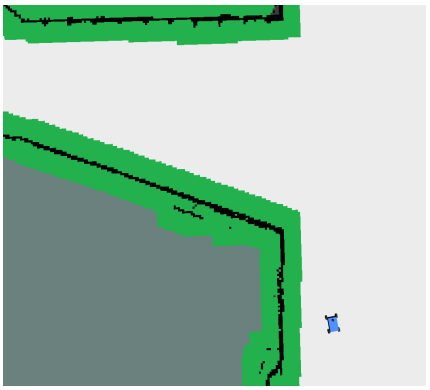

Abbildung 2. gepufferte Karte

III. RRT

Rapidly-exploring random tree (RRT) ist ein gutes Verfahren, um ein Weg durch ein schon erkanntes Terrain zu erzeugen. RRT startet hierbei bei einem Startknoten, welcher ebenfalls in unserem Fall die Position des Autos ist. Es wird jedem Iterationsschritt ein zufälliger Punkt q_{rand} erzeugt. Dann wird in dem bestehenden Graphen, der anfangs nur aus dem Startknoten besteht, nach dem Knoten $q_{nearest}$ gesucht, der am nächsten an q_{rand} liegt. Anschließend wird von $q_{nearest}$ eine festgelegte Schrittlänge in Richtung von q_{rand} gelenkt. Es wird also ein neuer Knoten q_{new} erzeugt der genau eine Schrittlänge von $q_{nearest}$ in Richtung von q_{rand} entfernt liegt. Wenn es kein Hindernis zwischen $q_{nearest}$ und q_{new} gibt, wird q_{new} dann dem Graphen hinzugefügt. Dies wird so lange wiederholt bis die maximale Anzahl an Iterationen erreicht ist, oder in unserem Fall bis das Ziel erreicht worden ist. Ist die Endbedingung erfüllt, wird der Pfad von dem Knoten, der am nächsten am Ziel liegt zurück verfolgt. RRT wird genauer umso mehr Iterationen und umso geringer die Schrittlänge definiert ist.

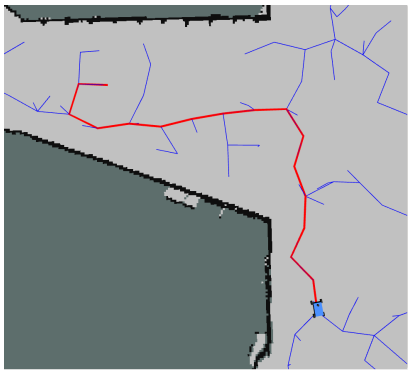

Abbildung 3. RRT

Umsetzung:

Nachdem wir die die Schwierigkeiten mit der Visualisierung von RRT überwältigt hatten, war die Implementierung des Algorithmus eigentlich kein großes Problem. Der schwierigste Teil war die Kollisions-Überprüfung, da wir hierfür die Karte aus der Simulation bzw. SLAM interpretieren mussten. In der Simulation selbst wird eine Kollisionsabfrage durchgeführt, um die Lidar-Messungen zu simulieren, also konnten wir das für unsere Kollisionsabfrage als Grundlage nehmen. Da es bei RRT viele Parameter gibt, die optimiert werden müssen, haben wir mit ROS-Parametern gearbeitet. Auf diese Weise mussten wir nicht jedes Mal das Programm neu kompilieren, wenn wir die Parameter ändern wollten.

IV. RRT*

Kurze Einführung in RRT*:
RRT* ist eine Erweiterung von RRT, bei der der komplette Suchbaum bei jedem neuen Knoten reevaluiert wird und wenn möglich so angepasst wird, sodass der Pfad immer so kurz wie möglich gehalten wird. Dies wird erreicht, indem nach jedem Hinzufügen eines Knotens q_{new} geprüft wird, ob der Pfad über einen anderen Knoten q_{near} innerhalb einer Nachbarschaft X_{near} um q_{new}, kürzer wäre, als über $q_{nearest}$. Außerdem wird danach auch noch überprüft, ob der Pfad von den Knoten innerhalb X_{near} über q_{new} eventuell kürzer ist als über den aktuellen Elternknoten. Falls der Weg von q_{near} über q_{new} kürzer ist, wird q_{new} als neuer Elternknoten von q_{near} eingetragen.
Mit RRT* ist es mit genügend vielen Knoten möglich, den kürzesten Pfad von einem Punkt zum anderen zu finden.

Umsetzung:

Um RRT* zu implementieren, mussten wir noch einige fehlende Funktionen, sowie die Kosten der jeweiligen Knoten hinzufügen.

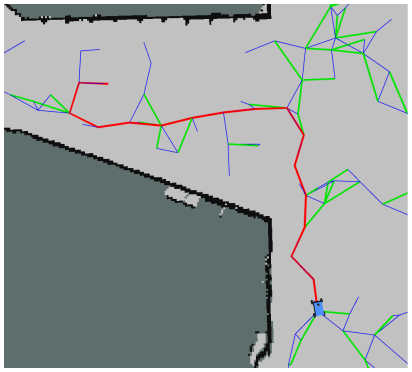

Abbildung 4. RRT*

Die Kosten eines Knotens berechnet sich aus der Distanz zu seinem Elternknoten, addiert mit dessen Kosten. Somit ist der kostengünstigste Pfad automatisch auch der kürzeste Pfad.

Anschließend definieren wir die Nachbarschaft, die wir als Menge der Knoten bezeichnen, die sich innerhalb eines bestimmten Radius befinden. Dieser Radius, berechnet sich jeweils aus

$$min(\eta \times (log(n)/n) \times 1/2, step\_length),$$

wobei

$$\eta >= 4(\mu(X_free)/\zeta_d)1/d,$$

mit
$d :=$ Dimension des Arbeitsraumes,
$\mu :=$das Lebesque Maß des Hindernisfreien Raums,
$\zeta_d :=$ das Volumen der Einheitskugel im d-dimensionalen Euklidischen Raum.

Dadurch wird der Radius mit zunehmender Anzahl an Knoten im Baum kleiner, bis er mindestens so groß wie die standardmäßige Schrittweite ist.
Nachdem ein Knoten hinzugefügt wurde, müssen nun alle Knoten in der Nachbarschaft überprüft werden, ob der Pfad über den neuen Knoten kürzer ist als über den bisherigen Elternknoten. Dies führt, aufgrund der momentanen Art der Speicherung des Baums, dazu, dass wir eine quadratische Laufzeitkomplexität erhalten, was leider nicht ohne weiteres gelöst werden kann.

Da jeweils bei der Erstellung des neuen Knotens, sowie bei der Reevaluierung der Nachbarschaft, die Kosten des Pfads kleinst möglich sind, wird bei genügend vielen Knoten auch der kürzeste Pfad gefunden.

Ein Ansatz wäre zum Beispiel, die Map in ausreichend kleine Abschnitte zu unterteilen, um dann die Knoten nach ihrer Position sortiert, zu speichern, um sofort auf bestimmte Knoten in bestimmten Bereichen der Map zuzugreifen. Eine weitere Möglichkeit besteht darin, die Knoten in einem 2-d Baum zu speichern und dann mittels Bereichssuche die Knoten innerhalb der Nachbarschaft zu ermitteln, was in $O(logn)$ möglich ist.

Um die Defizite unseres Algorithmus' auszugleichen, haben wir ein Bias implementiert. Dieses legt mit einer in den Parametern spezifizierten Wahrscheinlichkeit, den Punkt q_{rand} auf unser q_{goal}. Somit konzentrieren wir uns bei der Knotenerstellung auf die Zielrichtung und können dies bei gleichbleibender Knotenzahl schneller finden.

V. Pure Pursuit

Um die erstellte Trajektorie mit unserem Auto auch abfahren zu können, müssen wir einen Regler einbauen. Dazu nutzen wir einen Pure Pursuit Controller (PP). Dieser eignet sich für nicht glatte Trajektorien, sodass eine Glättung der errechneten Trajektorie aus RRT/RRT* nicht notwendig ist. Der Controller errechnet eine Kurve (Kreisausschnitt), die das Auto von seiner aktuellen Position auf eine Zielposition auf der Trajektorie zurückbringt. Die Zielposition wird auf der Trajektorie in einer definierten Entfernung, der *lookahead distance*, gesucht. Der Algorithmus iteriert ständig. Dabei sucht er die neue Zielposition und berechnet einen Kreisbogen (Kurve) dorthin. Der Kreisausschnitt wird tangential zur Fahrtrichtung an die Hinterachse des Autos angelegt und der Radius dieser so berechnet, dass die Zielposition auf dem Kreisausschnitt liegt. Somit muss die Orientierung des Autos nicht weiter beachtet werden. Da der Kreisbogen ständig neu berechnet wird, besteht keine Gefahr, dass das Auto u. U. in einem zu spitzen Winkel auf die Trajektorie trifft oder gar entgegen der Zielrichtung steht.

Umsetzung:

Auch bei Pure Pursuit war die Visualisierung ein großer Teil des Aufwandes. Wir haben die *lookahead distance* durch eine gründe Linie und den berechneten Kreis durch einen grauen Cylinder dargestellt. Da der Cylinder-Marker in rviz durch relativ grobe Ecken approximiert ist, entsteht bei der Visualisierung teilweise der Eindruck, dass der RRT-Pfad oder das Auto nicht direkt auf dem Wendekreis liegen. Dies liegt aber nur an der eingeschränkten Darstellung von runden Körpern und tritt auch nur auf wenn der Radius des Wendekreises sehr groß ist.

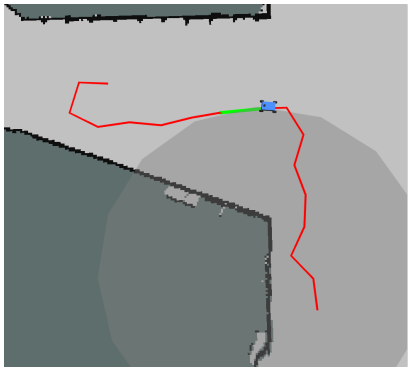

Abbildung 5. Pure Pursuit

Bei der Implementierung des Algorithmus war der komplizierteste Teil einen Punkt auf dem RRT-Pfad zu berechnen, der genau eine *lookahead distance* von dem Auto entfernt ist. Dafür iterieren wir erst durch alle Knoten auf dem Pfad und berechnen die Distanz zwischen dem Knoten und dem Auto bzw. die Differenz zwischen dieser Distanz und unserer erwünschten *lookahead distance*. Wenn diese Differenz das Vorzeichen wechselt, wissen wir, das der gesuchte Punkt zwischen dem aktuellen Knoten A und vorherigen Knoten$B$ liegt. Natürlich kann aus auch mehrere Punkte auf dem Graphen geben, die diese Voraussetzung erfüllen, also fangen wir am Ende vom Pfad mit dem Iterieren an um den Punkt zu finden der am nächsten am Ziel ist. Auf diese Weise wir also sichergestellt, dass das Auto den Pfad auch in die richtige Richtung abfährt. Nun wissen wir, dass der gesuchte Punkt auf der Geraden zwischen Knoten A und Knoten B liegt. Zusammen mit der Position des Autos ergibt sich das Dreieck in Abb. 6. Gesucht ist jetzt der Punkt P der genau eine *lookahead distance* l von C entfernt ist und auf $\overline{AB}$ liegt. γ kann man einfach über den Kosinussatz berechnen:

$$\gamma = arccos(\frac{b^2+c^2-a^2}{2bc}).$$

Die Distanz $c1$ von A zu P lässt sich wie folgt berechnen:

$$c_1 = bcos(\gamma) \pm \sqrt{b^2(cos(\gamma)^2 - b^2 + l^2}$$

Es gibt also auch hier zwei verschiedene Lösungen wobei oftmals eine der Lösungen negativ oder größer als $|AB|$ ist und somit nicht relevant ist. Auch in diesem Fall nehmen wir natürlich die Lösung die am weitesten fortgeschritten auf dem Pfad liegt, solange diese auf $\overline{AB}$ liegt. Mit $c1$ können wir anschließend natürlich einfach die Koordinaten von P berechnen. Wir können nun anhand von P,C und der Ausrichtung α des Autos den Winkel zwischen der aktuellen Ausrichtung und $\overline{CP}$ bestimmen:

$$\gamma = atan2((P.y - C.y), (P.x - C.x)) - \alpha.$$

Die Krümmung κ des Kreisbogens den wir für Pure Pursuit brauchen, lässt sich dann ebenfalls berechnen:

$$\kappa = \frac{2sin(\alpha)}{l}$$

Der Lenkwinkel ω den man braucht um eine Kurve mit der Krümmung κ zu fahren, kann man mithilfe des Abstand zwischen Vorder- und Hinterachse berechnen:

$$\omega = atan(\kappa * wheelbase)$$

Die Geschwindigkeit des Fahrzeugs ist bei uns proportional zum Winkel α, da das Fahrzeug langsamer Fahren soll wenn der RRT-Pfad eine scharfe Kurve beschreibt oder er stark von der RRT-Pfad abkommt. Die *lookahead distance* sollte eigentlich auch proportional zur Geschwindigkeit sein, aber in unserem Fall haben wir in der Regel die besten Ergebnisse mit einer konstanten *lookahead distance* erreicht.

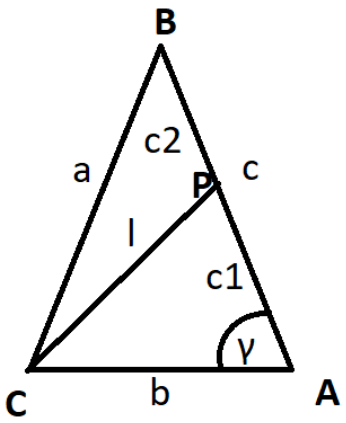

Abbildung 6. Berechnung vom Look-Ahead-Punkt

VI. Umsetzung ausserhalb der Simulation

Da der 2019-Skeleton-Code, den wir als Grundlage benutzt haben keine Möglichkeit bietet, um das Programm außerhalb der Simulation zu starten, mussten wir die erforderlichen Nodes für die Kommunikation mit Sensoren und Motoren aus dem älteren 2018-Skeleton übernehmen. Für den Pure Pursuit Controller benötigen wir außerdem die Position des Autos. Dafür benutzen wir das bereits vorhandene Vicon-System. Die Einbindung des Kamerasystems hat problemlos funktioniert, aber wir hatten kurzzeitig Verbindungsprobleme mit dem Vicon-Netzwerk. Damit die Transformation zwischen den verschiedenen Koordinatensystemen in ROS weiterhin funktioniert, starten wir auch weiterhin die Simulations-Node, wobei wir diese so umgeschrieben haben, dass sie zwischen einem Start in der Simulation und einem Start auf dem Auto unterscheidet und somit bei einem echten Start lediglich die Transformationen ausführt. Wir haben ein Launch-File geschrieben, was alle notwendigen Nodes für den Start auf dem Auto startet und die Parameter entsprechend setzt.

VII. Fazit

Pure Pursuit und RRT bzw. RRT* harmonieren sehr gut miteinander. Allerdings benötigt man, um einen möglichst glatten Pfad mit RRT* zu erstellen, sehr viele Punkte. Dies ist im momentanen Code jedoch nicht realistisch, da wir an verschiedenen Stellen den kompletten Baum durchsuchen. Um dies zu verhindern müsste der Baum in einer geeigneteren Speicherstruktur (z.B. k-d-Baum) implementiert werden.

Zudem benötigt man für die Ausführung eine Karte und eine aktuelle Position, welches für die Umsetzung in einem großen Maßstab eher schlecht geeignet ist.

VIII. KURZE ANLEITUNG

Um das Projekt auszuführen muss man nur das entsprechende Launch-File starten. Für den Start in der Simualation gibt es das Launch-File *simulator.launch* im Package *racecar_simulator* und für den Start auf dem Auto gibt es *real.launch* im gleichen Package. Um auch weiterhin unsere Visualisierung der Algorithmen in Rviz zu sehen, kann man ein ROS-Netzwerk über mehrere Computer aufbauen. Dafür müssen auf dem Auto die *$ROS_MASTER_URI* und *$ROS_IP* auf die IP des Autos gesetzt werden. Auf dem anderen Computer im gleichen Netzwerk muss dann entsprechend ebenfalls *$ROS_MASTER_URI* auf die IP des Autos gesetzt werden. Dann kann rviz auf diesem Computer mit der Konfigurationsdatei *simulator.rviz* (Package: *racecar_simulator*) gestartet werden und sollte nun alle Topics und Visualisierungen wie gewohnt anzeigen.

Um eine neue Karte mit Hector SLAM zu generieren, sollte man zunächst *teleop.launch* aus dem 2018-Skeleton ausführen. Das ist notwendig um die Lidar-Node und die Steuerung über den Controller zu starten. Mit dem 2019-Skeleton hatten wir hier leider Probleme mit den Koordianatentranformationen, die mit dem älteren Skeleton nicht aufgetreten sind. Anschließend kann man *tutorial.launch* aus dem *hector_slam_launch* package starten um eine Karte zu generiern. In diesem Launch-File kann auch eingestellt werden, ob es direkt anhand von den aktuellen Lidar-Daten generiert werden soll oder anhand von einer vorher aufgenommenen rosbag.

IX. CODE

https://github.com/JakobGreten/AutonomerUnfall

X. QUELLEN

Benutze Skeletons:
https://github.com/mlab-upenn/f110-fall2019-skeletons
https://github.com/cyphyhouse/f1tenth_code
https://github.com/mlab-upenn/f110-fall2018-skeletons
https://github.com/mlab-upenn/f110_rrt_skeleton
https://github.com/JakobGreten/AutonomerUnfall