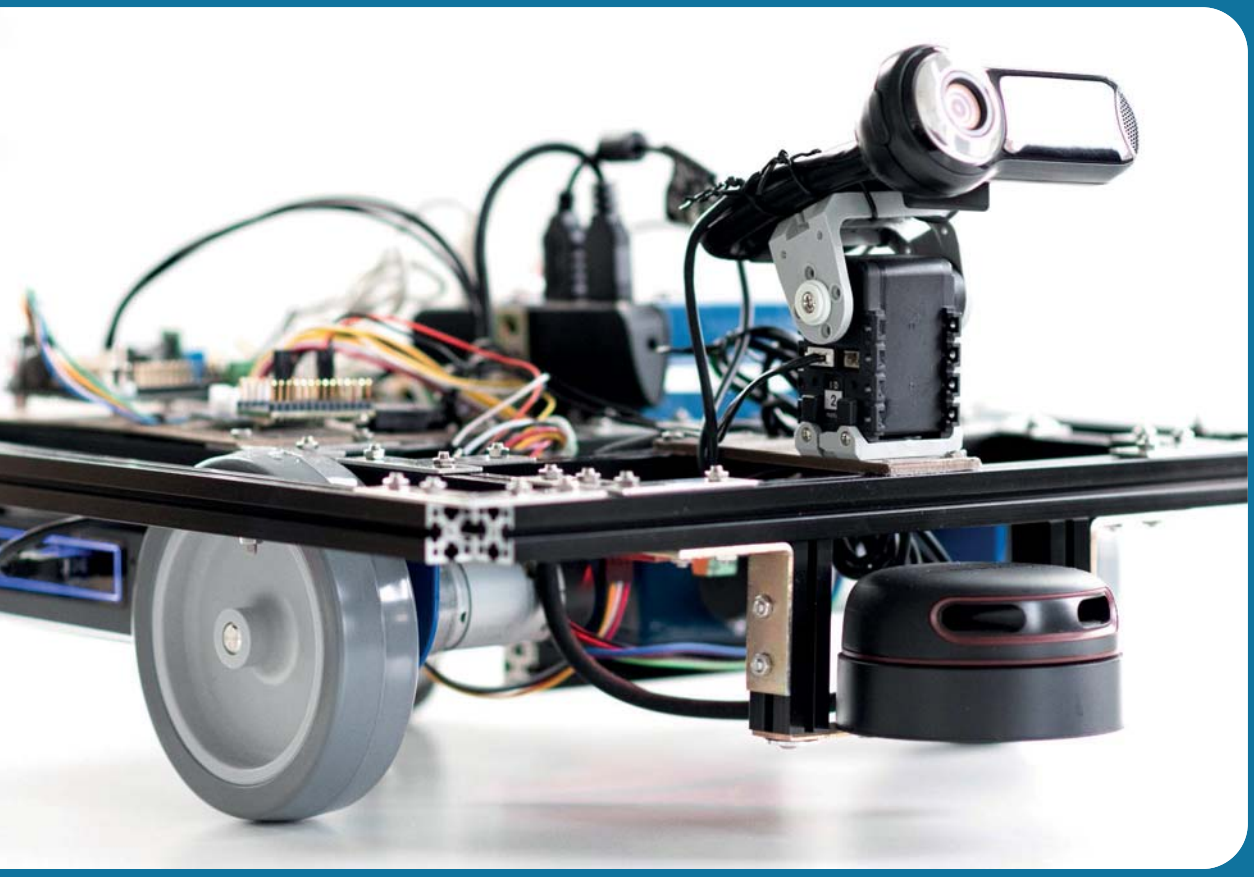
2 Roboter konstruieren und simulieren



In diesem Kapitel lernen wir Programme und Werkzeuge kennen, um Roboter in Simulationen und ROS verwenden zu können. Die Konstruktionsphase ist ein sehr frühes Stadium in der Entstehung eines Roboters. Deshalb nutzt man sie idealerweise zum Evaluieren von Prototypen. Wir konstruieren einen Roboter zunächst rein softwarebasiert. Anschließend wird der aktuelle Stand des Roboters im Simulator getestet. Gab es konstruktionsbedingte Mängel während der Simulation, setzen wir uns erneut mit der Konstruktion auseinander. Nach jeder Modifikation testen wir den Roboter im Simulator. Diesen Vorgang wiederholen wir so lange, bis wir der Meinung sind, dass der Roboter der Realität und den Bedingungen standhält, denen er später ausgesetzt sein wird. In einem Simulator führt man mit einem virtuellen Roboter Tests durch, die in der Realität große Schäden und hohe Kosten verursachen könnten. Simulationen sind aber nicht nur eine Testumgebung, um rechtzeitig Defizite der Hardware zu erkennen. Sie sind für Wissenschaftler und Forscher die optimale Umgebung, um Software zu evaluieren. Stellen Sie sich Tausende Roboterarme vor, die jeweils eine Kamera haben, um Objekte zu erkennen. Jeder Roboterarm arbeitet isoliert für sich und hat einen Heuhaufen vor sich liegen. Das Ziel ist, eine Nadel aus diesem Heuhaufen herauszuholen. Nun können Tausende unterschiedliche Algorithmen oder Verfahren getestet werden, um herauszufinden, welche die besten oder die schnellsten sind. Solche Tests würden in der Realität Unmengen an Kosten verursachen.

Mit ROS, der Simulationssoftware Gazebo und *RViz* haben wir die nötigen Werkzeuge zur Hand.

ROS-Roboter basieren auf *URDF*, dem *Universal Robot Description Format*. Gazebo basiert auf *SDF*, dem *Simulation Description Format*. Diese Unterschiede in den verwendeten Formaten stellen für uns kein Problem dar, da Gazebo auch *URDF*-Dateien lädt und verarbeitet. Zunächst muss also eine *URDF*-Datei erstellt werden, die unseren Roboter visuell und physikalisch repräsentiert.

In einer frühen Phase der Konstruktion dient *RViz* zur Visualisierung und Validierung *kinematischer Ketten*,[1] die wir mit *URDF*-Dateien beschreiben werden. Darüber hinaus können in *RViz* anhand interaktiver Marker Navigationsziele für Fahrzeuge oder Wegpunkte für *Endeffektoren*[2] eingegeben werden. Diese Navigationsziele oder Wegpunkte werden dann von unserem Roboter angesteuert.

Die Beispiele in diesem Buch basieren auf Gazebo in der Version 7, da es die zu ROS Kinetic kompatible Version ist und standardmäßig in Ubuntu 16 angeboten wird.

1. Eine *kinematische Kette* besteht aus Gliedern und Gelenken. Ein Gelenk, meist bestehend aus einem Servomotor, dreht sich um eine Achse. Jeweils eine unterschiedliche Drehachse kann als Freiheitsgrad betrachtet werden. Es sind insgesamt sechs Freiheitsgrade möglich – oben/unten, rechts/links, vor/zurück, drehen, neigen und rollen.
2. Am Ende einer kinematischen Kette befindet sich der *Endeffektor* – ein Greifer oder eine Roboterhand.

Worauf hier nicht eingegangen wird, sind der Entwurf und die Konstruktion von Roboterteilen bzw. Robotern in CAD-Programmen. Wenn für bestimmte Arbeitsschritte externe CAD-Software benötigt wird, werden wir uns mit Blender und FreeCAD behelfen. Beide Programme sind kostenlos und es existieren ausreichend Video-Tutorials im Internet.

In den nächsten Kapiteln werden wir uns mit folgenden Themen auf dem Entwicklungsrechner beschäftigen:

- Gazebo und Simulationen
- RViz und kinematische Strukturen
- FreeCAD und Formatkonvertierungen
- Blender und 3D-Modelle
- URDF und das Roboter-Modell

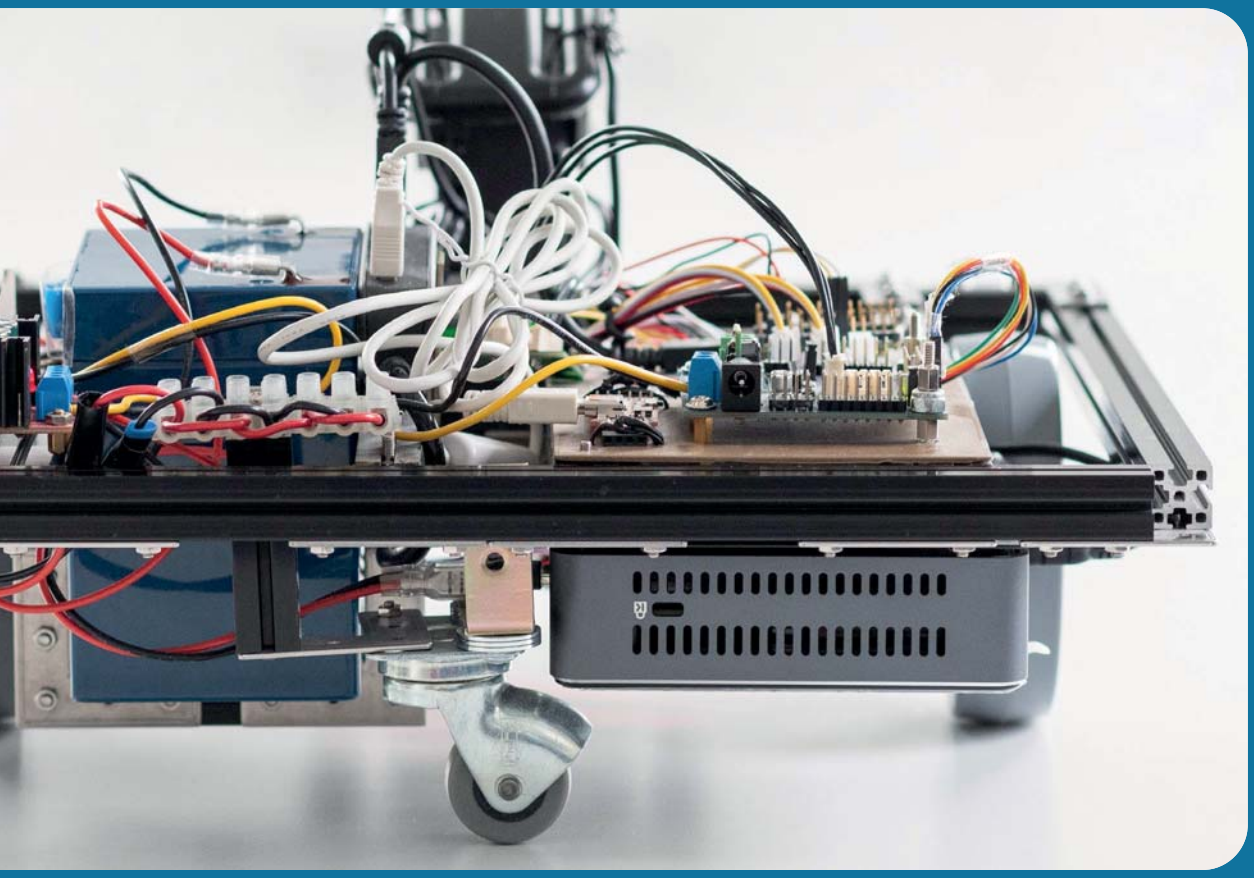
2.1 Gazebo

Simulationen dienen auch der Qualitätsprüfung. Bevor ein Roboter tatsächlich gebaut wird, durchläuft der virtuelle Roboter verschiedene Tests in einer simulierten Umgebung. Da in einer Simulation nichts kaputt gehen kann, können die Tests so oft wie nötig ausgeführt werden. Eine Simulation ist ein mathematisch berechnetes Abbild der Realität. Und genau hier liegt auch ein Defizit heutiger Simulationen. Möchte man die Realität detailgetreu und mit allen physikalischen Eigenschaften simulieren, bräuchte man eine enorme Rechenkapazität, die es aktuell so noch nicht gibt. Man denke an das Earth Simulator Project, welches globale Klimamodelle simuliert und dessen Rechenzentrum so groß ist wie eine Sporthalle. So ist es eben noch nicht möglich, Regen, Schnee oder Nebel sowie Audio und Flüssigkeiten in Gazebo zu simulieren.

> *Ein Simulator in der Konstruktionsphase hilft, Konstruktionsfehler früh zu erkennen.*

Mittlerweile wird Gazebo weltweit in Wettkämpfen in der *RoboCup*-Simulations-Liga sowie der *Virtual Robotics Challenge* eingesetzt und für die Mars-Mission veranstaltet die NASA die *Space Robotics Challenge*. Die benötigten Roboter gibt es als steuerbare 3D-Modelle in Gazebo unter dem Bedienfeld *Insert*. Standardmäßig werden Benutzermodelle im Heimatverzeichnis des aktuellen Benutzers gespeichert: *~/.gazebo/models*.

3 Roboter-projekt A



Im Roboterprojekt A entsteht ein leistungsstarker Roboter mit hoher Rechenkapazität und langer Akkulaufzeit. Ein kraftvolles Antriebssystem sorgt für ausreichend Schub bei Traglasten bis 5 kg. Der flache Aufbau ermöglicht die Erweiterung mit einem Roboterarm oder um einfach nur Güter zu transportieren. Eigene Modifikationen oder Prototypen sollen mit diesem Modell einfach und schnell realisierbar sein.

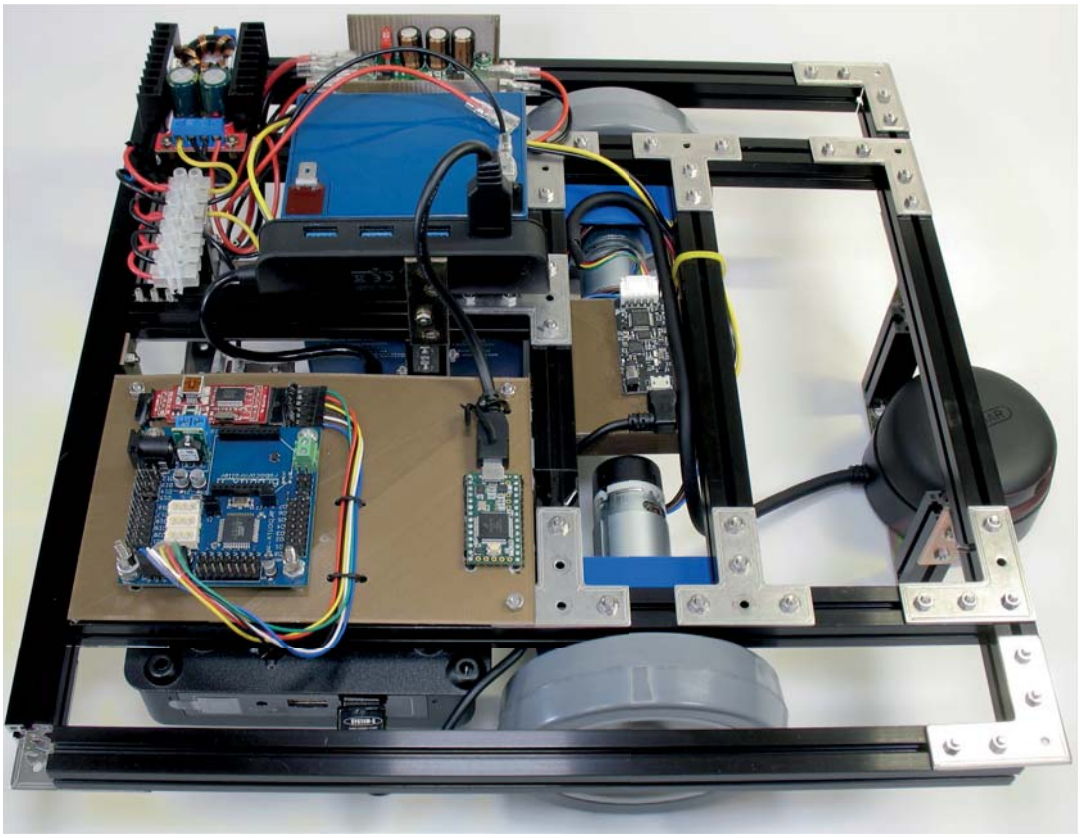


Abb. 3–1 Roboterprojekt A

3.1 Ziel

Oft heißt es »Man muss klein anfangen«. Zugegeben, mein erster Roboter war eine kleine mobile Basis mit drei Infrarotsensoren und einem *ATMega32*. Damit kann man insektenartige Verhalten programmieren, aber mehr nicht. Mit ROS haben wir uns bereits entschieden, etwas »Größeres« zu erschaffen. Nun benötigen wir ein passendes Gefäß, um diese Möglichkeiten ausschöpfen zu können. Ein Blick auf die Roboter,[1] die für ROS gebaut wurden oder ROS verwenden, kann hilfreich bei der Findung eines Konzepts für den eigenen Roboter sein. Anders als organische Lebewesen, die als kleine Zellen beginnen und durch Zellteilung auswachsen, befinden sich Roboter, die erschaffen wurden, am Anfang und am Ende ihrer Lebenszeit stets in einem ähnlichen Zustand. Wenn man sich also für ein Konzept entscheidet, sind die Anforderungen für heute und für die Zukunft mit eingeplant. Das Ziel dieses Projekts soll sein, dass der Roboter alle Möglichkeiten von ROS ausnutzen kann und gleichzeitig unsere eigenen Programmierideen performant ausführt. Beginnen wir mit den Primäranforderungen, die dafür von unserem Roboter erfüllt werden müssen.

Der Roboter soll kollisionsfrei navigieren. Das beinhaltet die Erkennung von Hindernissen und Fortbewegung auf einer planen Fläche, während gleichzeitig eine Karte von der Umgebung erstellt wird. Die so erstellte Karte kann dann gespeichert und jederzeit geladen werden, um die darauffolgenden Navigationen mit einer vorhandenen Karte zu beginnen. Heutzutage verwendet man meist einen *LIDAR (engl. Light Detection And Ranging)*[2] oder ein ähnliches System, um Karten zu generieren. Zur Fortbewegung wird ein Antriebssystem benötigt. Diese Rolle übernimmt ein Differenzialantrieb, der aus zwei separat angetriebenen Rädern auf einer Achse besteht. Zudem benötigt man ein oder zwei passiv mitlaufende Möbelroller. Die Disziplin der Kartenerstellung und der kollisionsfreien Navigation ist ein Hauptmerkmal von ROS. Diese Funktionalität ist von Haus aus mit dabei.

Eine Gesichtserkennung soll den Roboter für die Interaktion mit Menschen sensibilisieren. Im Idealfall sollen die erkannten Gesichter gespeichert und gelernt werden, damit in Zukunft auf immer wiederkehrende Gesichter bzw. Personen individuell reagiert werden kann. Hierfür ist eine handelsübliche USB-Kamera ausreichend. Befestigt man die Kamera zusätzlich auf einem motorisierten Gelenk, kann sie dynamisch ausgerichtet werden. Für die Gesichtserkennung gibt es bereits fertige Bibliotheken und Modelle, nur die Motorsteuerung bedarf etwas Programmierarbeit.

1. Viele Roboter, die ROS verwenden, finden sich im Internet unter *https://robots.ros.org/*.
2. Ein LIDAR ist ein laserbasiertes Messystem, um die Entfernung zu einem Gegenstand zu messen. Dabei wird ein Lichtimpuls ausgesendet, der von der Umgebung reflektiert wird. Die Reflexion kann gemessen und zur Ermittlung der Distanz verwendet werden. Ein ähnliches Prinzip der Reflexionsmessung existiert unter dem Begriff *Radar (engl. Radio Detection And Ranging)*.

Vorab legen wir die Betriebspannung fest, mit der wir unseren Roboter betreiben wollen. Eine 240-V-Spannung ist zu hoch, während eine 5-V-Spannungsquelle nicht ausreichend Energie einem Motor zur Verfügung stellen kann. Am einfachsten bestimmen wir die benötigte Spannung anhand der Endverbraucher, die wir einsetzen werden. Ein Mini-PC läuft erst ab 12 V, dann kommen die Motoren, die ebenso auf 12 V angewiesen sind. Sollte ein Verbraucher weniger benötigen, kann man mit den 5 V aus der USB-Buchse arbeiten. Eine andere Möglichkeit ist ein Pegelwandler, der aus einer Eingangsspannung die gewünschte Ausgangsspannung generiert. Für die meisten Anforderungen wird uns eine 12-V-Spannung ausreichen. Damit die Spannung konstant bleibt, bauen wir an geeigneter Stelle einen einstellbaren Aufwärtswandler (engl. *Booster*) ein, und lassen ihn auf einer Ausgangsspannung von 12 V, damit der Computer oder die Motoren bei sinkender Batteriekapazität nicht ausfallen oder an Kraft verlieren.

Wenn alle Stromverbraucher unter Last arbeiten, soll der Roboter mindestens eine Stunde autark sein, bevor er wieder an die Ladestation muss. Dieser Anforderung genügt ein 12-V-Batterie mit ca. 5000 mAh. Bei der Wahl des Akkus steht die Sicherheit an erster Stelle. Der Roboter wird in geschlossenen Räumen eingesetzt, wo auch Menschen anwesend sein können. Ist der Roboter allein zuhause, darf sein Akku nicht in Brand geraten, da niemand das Feuer löschen könnte.

Die bereits genannten Anforderungen verlangen ausreichend Rechenkapazität, die wir mit einem Mini-PC auf x86-Basis bereitstellen werden. Nach oben gibt es bekanntlich keine Grenzen, wenn es um Rechenleistung geht. Wir kalkulieren bei der Wahl des PCs die gleichzeitig laufenden Programme und zusätzlich noch etwas Kapazität für Hintergrundberechnungen ein. Wenn später weitere Anforderungen hinzukommen, sollten auch für diese noch ausreichend Resourcen berücksichtigt werden. Ein Beispiel wäre der Anbau eines Roboterarms und die dafür anfallenden Kinematikberechnungen.

Damit wir nicht einen riesigen Klotz am Bein haben, während wir die Teile bestellen, machen wir für unseren Roboter eine quadratische Flächenvorgabe von 30 × 30 cm. Eine Abweichung nach oben oder nach unten sollte nicht mehr als 5 cm betragen. Die Flächenvorgabe bietet eine ausgewogene Balance zwischen Größe und Stabilität. Ein kompakter Roboter, der in die Höhe gebaut wird, kippt schneller um als einer, der breiter angelegt ist. Als Berechnungsgrundlage kann man eine Wohnungstür verwenden. Durch eine Tür sollten ein Mensch und ein Roboter bequem gleichzeitig durchpassen.

Das Ganze soll schließlich zwischen 1200,– und 1600,– € kosten. Nicht einkalkuliert sind Werkzeuge, wie Schraubenzieher, Multimeter und Bohrmaschine. Diese Dinge verbleiben nicht am Roboter und wurden deshalb im Gesamtpreis nicht mitgerechnet.

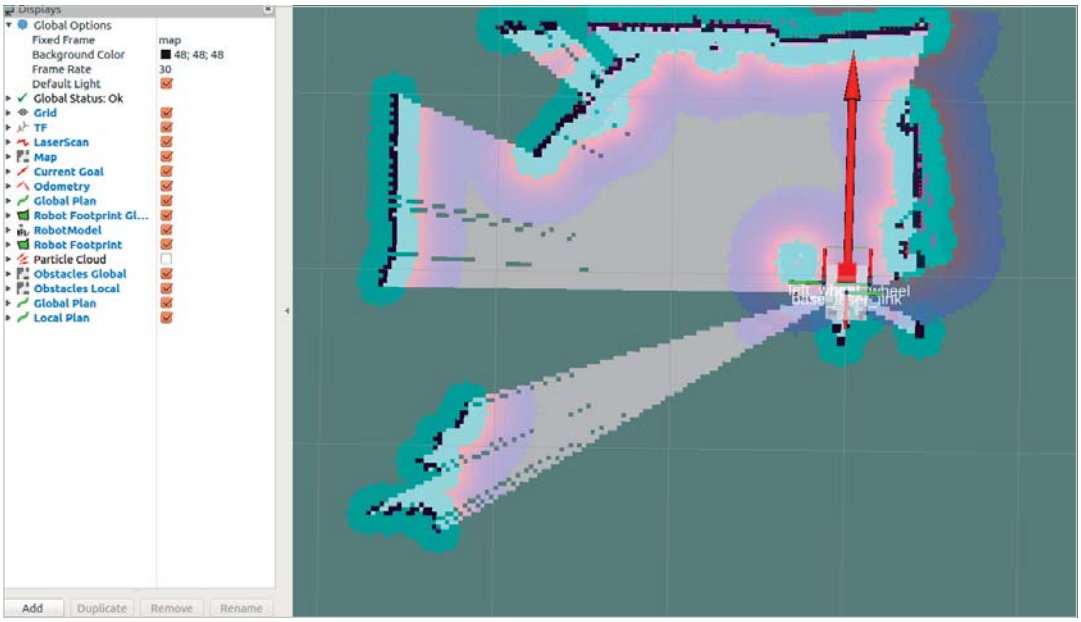


Abb. 5–2 RViz-Sensordaten von einem entfernten Rechner

5.3 Hallo Welt

Das erste ROS-Programm soll so klein und überschaubar wie möglich sein. Bevor man ein Python- oder C++-Programm in ROS schreiben kann, müssen die nötigen Rahmenbedingungen erfüllt werden. Benötigt wird selbstverständlich erst mal eine ROS-Installation. Nach der Installation wird ein Arbeitsbereich erstellt und mit dem Befehl *catkin_make* initialisiert. Im *src*-Verzeichnis des *catkin*-Arbeitsbereichs kann schließlich ein ROS-Paket mit dem Befehl *catkin_create_pkg* generiert werden. In dem so entstandenen Paketverzeichnis befindet sich das Grundgerüst eines jeden ROS-Pakets, darunter auch das Verzeichnis *src* für eigene C++-Programme und Python-Module. Bis zum Erstellen eines Arbeitsbereichs wurde bereits alles in Abschnitt 1.1 beschrieben. Wir knüpfen in diesem Kapitel daran an und beginnen mit der Erstellung eines neuen ROS-Pakets. Danach schreiben wir Programme mit Python und mit C++. Sie müssen sich nicht auf eine dieser beiden Sprachen für immer festlegen. Sie können einen Teil mit der einen und einen anderen Teil mit der anderen Sprache programmieren. Einige Empfehlungen gehen davon aus, dass man in der *Prototyping*-Phase alles in Python schreibt und erst später für die *Produktiv*-Phase alles in C++ umschreibt. In anderen Empfehlungen werden nur performancekritische Bereiche in C++ programmiert und der Rest in Python. Sie sollten am besten selbst

herausfinden, was die beste Lösung ist. Oft ist das reine Geschmackssache und das ist gut so.

Die am Anfang dieses Kapitels beschriebenen Schritte sind ein kleiner Auszug der Prämissen, die für ein ROS-Programm benötigt werden. Im Detail sind da beispielsweise noch die Dateien *CMakeLists.txt* sowie *package.xml*. Ohne diese Dateien und deren sachgemäße Konfiguration wird kein ROS-Programm starten. Als Einsteiger in ROS kann dies eine echte Herausforderung sein. Umso wichtiger ist es, mit einem einfachen Beispiel zu beginnen. Testen Sie mit dem nächsten Befehl, ob der Arbeitsbereich des aktuellen Benutzers erreichbar ist.

```
roscd
```

Wenn Sie nicht in einem Benutzerverzeichnis landen, sondern in einem Systemverzeichnis, wo ROS installiert ist, dann ist entweder die Umgebungsvariable *$ROS_PACKAGE_PATH* falsch konfiguriert oder es existiert gar kein *catkin*-Arbeitsbereich im Heimatverzeichnis des Benutzers. In Abschnitt 1.1.8 sind Arbeitsbereiche beschrieben.

Das korrekte Zielverzeichnis von *roscd* ist *~/catkin_ws/devel*. Löst man die Tilde auf, dann lautet der absolute Pfad */home/<benutzername>/catkin_ws/devel*, wobei Sie *<benutzername>* durch den Benutzernamen des angemeldeten Benutzers ersetzen. Wechseln Sie nun aus dem *devel*-Verzeichnis eine Verzeichnisebene nach oben, sodass sich das *src*-Verzeichnis unter Ihnen befindet. Anschließend steigen Sie in das *src*-Verzeichnis ab. Dieses Auf und Ab könnte man sich durch ein *Alias* in der *.bashrc* sparen. Ich erstelle mir dort gerne Pseudonyme für *roscd*, *catkin_make* und alles weitere, was sonst an Aufgaben wiederkehrt. Alternativ wechselt man mit *cd* direkt in das gewünschte *src*-Verzeichnis. Der folgende Alias führt ohne Umwege in das Verzeichnis *src* des Arbeitsbereichs.

```
alias rcd='cd ~/catkin_ws/src'
```

Im *src*-Verzeichnis angekommen, erstellen wir mit dem folgenden Befehl ein neues ROS-Paket:

```
catkin_create_pkg hallo_welt std_msgs rospy roscpp
```

Die Syntax von *catkin_create_pkg* erwartet als erstes Argument einen eigenen ROS-Paketnamen. Optional können Abhängigkeiten zu anderen ROS-Paketen angegeben werden. Die Paketabhängigkeiten müssen nach dem eigens definierten Paketnamen von Leerstellen getrennt angehängt werden. Die Abhängigkeiten können nachträglich in den generierten Dateien *CMakeLists.txt* und *package.xml* ergänzt oder gelöscht werden. Es ist also kein Problem, wenn einige Abhängigkeiten später hinzukommen oder nicht mehr verwendet werden. Die im vorigen Befehl angehängten optionalen Abhängikgeiten referenzieren Standardbibliotheken, die wir zum Programmieren mit