Poster:

Titel des Projektes

GPS auf Rädern

Bearbeitungszeitraum (das jeweilige Semester, in dem das Praktikum absolviert wurde)

Wintersemester 2014/2015

Name und Studienfächer der Studierenden

Jannis Andrija Schnitzer - Physik

Jonathan Förste - Physik

Gloria Feher - Angewandte Informatik

Betreuer und Supervisor

Prof. Dr. Katja Mombaur, Gero Plettenberg

Aufgabenstellung des Projektes

Das Ziel des Projektes ist es, ein Modellauto mittels eines Raspberry Pis und GPS-Moduls zu autonomer Geländefahrt zu befähigen. Nach Erhalt von entsprechenden Zielkoordinaten soll der Roboter mithilfe von Modul und externen Sensoren seinen Weg auch durch schwieriges Gelände finden und Hindernissen ausweichen können.

Ergebnis

Hardware:

- RC Auto (Modell Reely Rhino III Brushed)

- Maßstab: 1:10

- Allrad-Buggy mit 540er Elektromotor

- MC 4519 Lenkservo

- Raspberry Pi (Modell B)

- 700 MHz ARM1176JZF-S core CPU

- 512 MB RAM

- 8 x GPIO

- UART

- I2C bus

- Ultraschall-Sensoren (Modell HC-SR04)

- VCC +5V, GND 0V

- Distanzmessung: 2 – 400 cm

- Messintervall: 20ms à max. 50 Messungen/Sekunde

- Messwinkel: 15°

- GPS-Modul mit eingebautem Kompass (Modell ublox LEA-6H)

- VCC +5V; GND 0V

- Ca. 2.5 m Genauigkeit

Software:

- Raspbian

- gpsd library

- RPI.GPIO library

- Selbstentwickelter Code für:

- GPS-Datenverarbeitung und Routenplanung

- Ansteuerung des Motorreglers und Lenkservos des Autos

- Distanzmessung

- Hinderniserkennung und -verarbeitung

Materialien:

- Plexiglas

- Schrauben

- Jumper-Kabel

- Platinen

Funktionsweise:

Die Funktionalität des Roboters lässt sich in vier größere Abschnitte gliedern: Die Hardware-Ansteuerung, welche Schnittstellen für die Verwendung der einzelnen Komponenten bietet, die Navigationsroutine, welche berechnet, wie der Roboter fahren muss um an die Zielkoordinaten zu gelangen, die Hinderniserkennung, welche die Daten der Ultraschallsensoren verarbeitet, Hindernisse erkennt und ihnen ausweicht und die Hauptroutine, welche alle Software-Komponenten miteinander verbindet.

Hardware-Ansteuerung:

Lenkservo und Fahrtmotor werden über den Raspberry Pi mithilfe von PWM-Signalen (Pulse-Width-Modulation) angesteuert. Hierbei gibt die Länge des gesendeten Signals an, wie die Motoren reagieren (siehe Exemplarische PWM-Signale und Reaktionen der Motoren).

Das GPS-Modul ist über UART (Universal Asynchronous Receiver Transmitter; serielle Schnittstelle) mit dem Pi verbunden. Der eingebaute Kompass wird über I2C (Inter-Integrated Circuit; serieller Datenbus) angesteuert. Hierfür werden jeweils die designierten UART und I2C Pins des Raspberry Pis sowie die gpsd Library verwendet.

Die Ultraschallsensoren werden über eine kleine Schaltung, die garantieren soll, dass der Pi keine Signale über 3.3V erhält, mit dem Minicomputer verbunden (siehe Schaltplan: Verbindung der Ultraschallsensoren mit dem Raspberry Pi). Um die Distanz zu messen, wird ein Ultraschallsignal gesendet (trig). Dann wird die Zeit gemessen, die vergangen ist bis das Signal zurückgeworfen wurde (echo). Die vergangene Zeit wird mit der Schallgeschwindigkeit (343 m/s) multipliziert und durch zwei geteilt, da die Strecke zwei mal zurückgelegt wurde. Somit erhält man die Distanz zum nächsten Objekt.

Navigation:

++++ Bitte ausführen ++++

Hinderniserkennung:

++++ Bitte ausführen ++++

Hauptroutine:

Zunächst wird überprüft, ob die über die Kommandozeile beim Starten des Programms eingegebenen Werte gültige Eingaben sind. Sind die Eingaben nicht von der Form "Zahl\_1 Zahl\_2", so beendet das Programm.

Als nächstes muss auf ein GPS-Signal gewartet werden. Sobald dieses vorliegt wird in eine Endlosschleife eingetreten und zunächst überprüft, ob das Auto gerade stillsteht.

Ist dies der Fall, so steht es noch am Anfang seiner Reise. Daher muss zunächst die Orientierung bestimmt werden, damit der Navigator den Kurs berechnen kann.

Dann wird überprüft ob das Auto bereits an seinem Ziel angekommen ist. Falls dem so ist, hält es an und das Programm bricht aus der Schleife aus.

Ist es nicht an seinem Ziel, wird überprüft ob sich das Auto auf dem richtigen Kurs befindet und gegebenenfalls auf den korrekten Kurs gebracht.

Die letzte if-Anweisung prüft, ob das Auto gerade fährt - und wenn dem so ist, speichert es seine Position in die Liste der letzten Positionen des GPSTrackers und prüft auf Hindernisse.

Bilder des Roboters, Visualisierung o. Ä.

- 3/4 Ansicht

- Bild direkt von oben

- PWM-Bild „Exemplarische PWM-Signale und Reaktionen der Motoren“

- Schaltplan: Verbindung der Ultraschallsensoren mit dem Raspberry Pi

- Zustandsdiagramm main.py

Webpage:

Titel des Projektes

GPS auf Rädern

Bearbeitungszeitraum (das jeweilige Semester, in dem das Praktikum absolviert wurde)

Wintersemester 2014/2015

Name und Studienfächer der Studierenden

Jannis Andrija Schnitzer - Physik

Jonathan Förste - Physik

Gloria Feher - Angewandte Informatik

Betreuer und Supervisor

Prof. Dr. Katja Mombaur, Gero Plettenberg

Aufgabenstellung des Projektes

Das Ziel des Projektes ist es, ein Modellauto mittels eines Raspberry Pis und GPS-Moduls zu autonomer Geländefahrt zu befähigen. Nach Erhalt von entsprechenden Zielkoordinaten soll der Roboter mithilfe von Modul und externen Sensoren seinen Weg auch durch schwieriges Gelände finden und Hindernissen ausweichen können.

Technische Details:

- RC Auto (Modell Reely Rhino III Brushed)

- Maßstab: 1:10

- Allrad-Buggy mit 540er Elektromotor

- MC 4519 Lenkservo

- Raspberry Pi (Modell B)

- 700 MHz ARM1176JZF-S core CPU

- 512 MB RAM

- 8 x GPIO

- UART

- I2C bus

- Ultraschall-Sensoren (Modell HC-SR04)

- VCC +5V, GND 0V

- Distanzmessung: 2 – 400 cm

- Messintervall: 20ms à max. 50 Messungen/Sekunde

- Messwinkel: 15°

- GPS-Modul mit eingebautem Kompass (Modell ublox LEA-6H)

- VCC +5V; GND 0V

- Ca. 2.5 m Genauigkeit

Software:

- Raspbian

- gpsd library

- RPI.GPIO library

- Selbstentwickelter Code für:

- GPS-Datenverarbeitung und Routenplanung

- Ansteuerung des Motorreglers und Lenkservos des Autos

- Distanzmessung

- Hinderniserkennung und -verarbeitung

Ergebnis mit Projektverlauf

Am Ende des Projektes steht ein funktionsfähiger Roboter, welcher der Aufgabenstellung entsprechend autonom seinen Weg zu eingegebenen GPS-Koordinaten findet ohne dabei gegen Hindernisse zu fahren.

Phase 1: Planung

Nach intensiver Recherche und ausführlichen Teambesprechungen wurde die Entscheidung gefällt, den Roboter so modular wie möglich aufzubauen:

Die Ansteuerung des Motorreglers und des Lenkservos sollte ein Mikrocontroller übernehmen.

Auf der nächsthöheren Schicht, sollte der Raspberry Pi liegen, welcher alle komplexeren Berechnungen, wie die Routenplanung, den Ausweichalgorithmus für Hindernisse und die Verarbeitung der Inputdaten der unterschiedlichen Sensoren durchführt.

Auf diese Weise wären einzelne Teile leicht austauschbar und böten so viele Möglichkeiten zur Weiterentwicklung des Roboters.

Hardware:

RC Auto:

Bei der Auswahl des RC Autos wurden Kriterien wie Größe und Robustheit besonders beachtet. Es war klar, dass das Auto genug Platz bieten müsse um den Raspberry Pi, das GPS-Modul und die Distanzmessungssensoren gut unterzubringen. Eine Stoßstange und gute Federung waren wichtig, um dem Roboter so viel Sicherheit wie möglich zu bieten.

Mikrocontroller:

Vorgabe war, die hauseigene Mikrocontroller-Platine des Robotiklabors der Universität Heidelberg zu nutzen. Diese wurde anhand der Lötanleitung zusammengebaut und mit einem ATmega168 bestückt.

Distanzmessungssensoren:

Um eine Distanz zu messen, können Infrarot- oder Ultraschallsensoren verwendet werden. Aufgrund der größeren Reichweite von Ultraschallsensoren fiel die Wahl auf diese, speziell auf das Modell HC-SR04 welches sich durch gute Bewertungen auszeichnet und durch seinen geringen Preis sogar in mehreren Instanzen eingesetzt werden kann, ohne das Budget zu sprengen. Zwei Ultraschallsensoren wurden für ausreichend befunden.

GPS-Modul:

Es handelt sich um das Modell ublox LEA-6H, welches nicht nur GPS-Daten liefert sondern auch einen eingebauten Kompass hat. Dieses wurde von den Betreuern des Praktikums ausgewählt.

Software:

Die Verwendung des Betriebssystems Raspbian für den Raspberry Pi war die einzige Vorgabe bezüglich der Software, die gemacht wurde.

Für die benötigten Libraries zur Ansteuerung der GPIO-Pins des Raspberry Pis und des GPS-Moduls fiel die Wahl auf die viel-verwendeten Libraries RPi.GPIO und gpsd.

Der selbst zu entwickelnde Code wurde in Sinnabschnitte (Hardwareansteuerung/-kommunikation, Routenplanung, Ausweichroutine) untergliedert und auf die unterschiedlichen Teammitglieder verteilt, wobei immer wieder Rücksprache gehalten wurde und der Code gegenseitig korrigiert wurde.

Für die Kommunikation zwischen Raspberry Pi und ATmega, wurde ein eigenes Kommunikationsprotokoll in mehreren Prototypen entworfen.

Phase 2: Verbindung der unterschiedlichen Komponenten

Die ersten Aufgaben bestanden darin, sich mit der Elektronik vertraut zu machen und zu lernen, wie einzelne Komponenten (z.B. die Servos des Autos) angesteuert werden können.

Servomotoren werden über PWM (Pulse-Width-Modulation) Signale gesteuert. Hierbei gibt die Länge des Signals vor, wie sich der Motor bewegt. Etwa alle 20 ms erwartet der Motor einen Input, dieser ist typischerweise zwischen 1 und 2 ms lang.

Um die genauen Werte für den Motor und den Lenkservo des Autos zu bestimmen, wurden die von der Fernbedienung übermittelten Signale mit einem Oszilloskop gemessen.

Exemplarische PWM-Signale und Reaktionen der Motoren

|  |  |  |
| --- | --- | --- |
| **PWM-Signallänge** | **Reaktion Lenkservo** | **Reaktion Fahrtmotor** |
| **1000** | --- | Vollgas rückwärts |
| **1170** | Max. links | Rückwärts |
| **1500** | Nullposition | Ausgeschaltet |
| **1800** | Max. rechts | Vorwärts |
| **1952** | --- | Vollgas vorwärts |

Messergebnisse

Die oben dargestellten Messergebnisse wurden dann in Code gegossen, um abstrahierende Fahrfunktionen zu implementieren.

Diese wurden zunächst auf dem ATmega in der Programmiersprache C unter Verwendung des Basis-Codes des Robotiklabors umgesetzt und funktionierten einwandfrei.

Um den Fahrtregler und damit den Fahrtmotor und den Lenkservo anzusteuern, wurde der ATmega über die Antennenbuchse mit dem Fahrtregler verbunden.

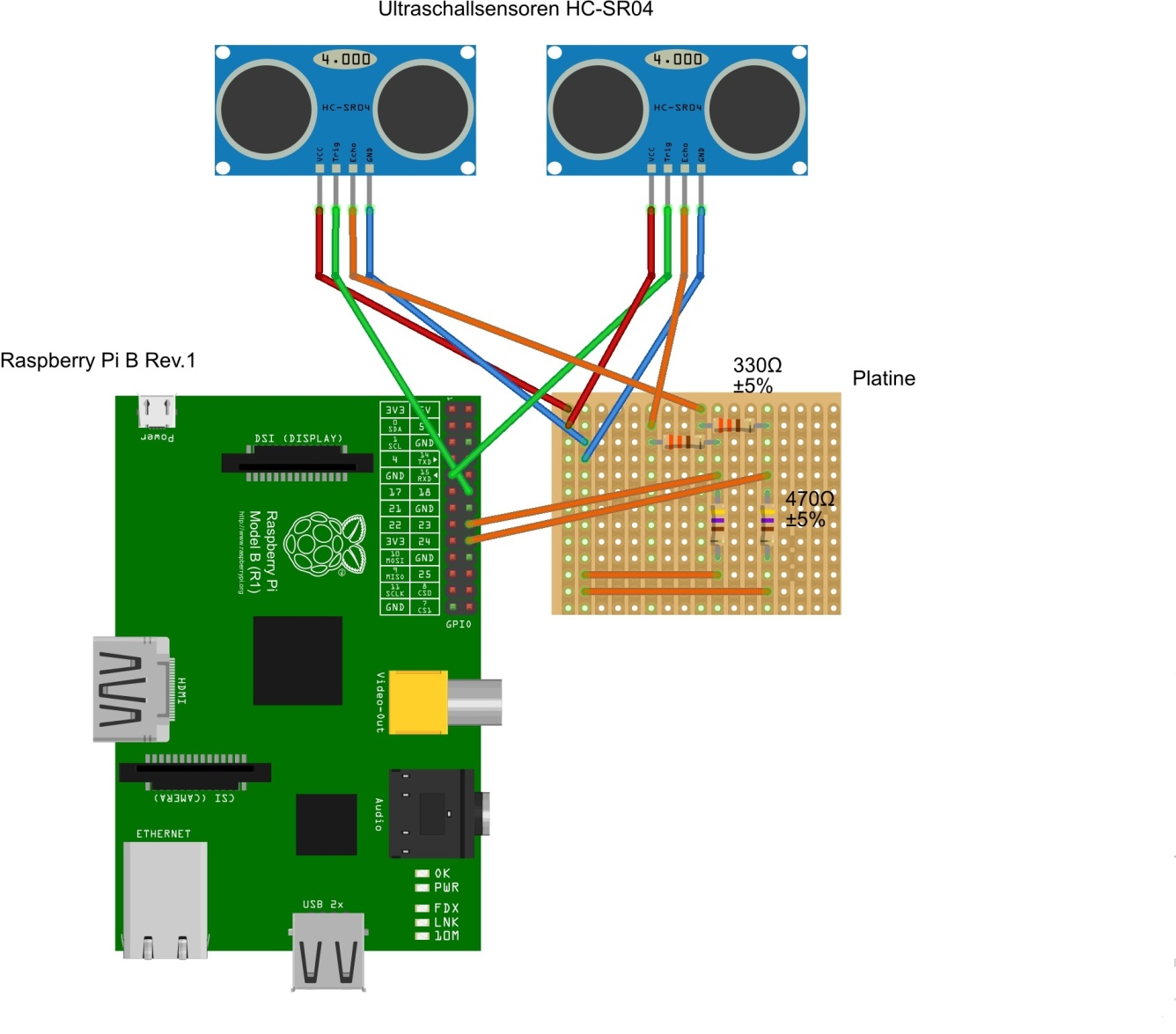
((Grafik???))

Das Kernstück des Roboters ist der Raspberry Pi und mit diesem mussten alle anderen Komponenten verbunden werden. Die designierten UART- und I²C-Pins des Raspberry Pis wurden verwendet um den GPS-Chip bzw. den Kompass anzuschließen.

Da jedoch auch die Verbindung zwischen Raspberry und ATmega über UART hergestellt werden musste, wurden zwei GPIO-Pins des Pis verwendet um eine weitere UART-Schnittstelle zu bitbangen (über Software zu emulieren). Hierzu kam die PIGPIO Library zum Einsatz. Über das selbstgeschriebene Kommunikationsprotokoll sollten nun Pi und ATmega kommunizieren können, jedoch gab es trotz funktionierender Kommunikationsschnittstelle Komplikationen mit dem Mikrocontroller, die in sinnvoller Zeit nicht überwunden werden konnten. So musste der ursprüngliche Entwurf überarbeitet werden.

Der Mikrocontroller wurde aus dem Aufbau entfernt, die Fahrfunktionen mussten auf dem Raspberry Pi in Python neu implementiert werden, und die PWM-Signale, die der ATmega zuvor generiert hatte, mussten nun durch den Pi erzeugt werden. Für letzeres wurde die RPi.GPIO Library verwendet, welche Funktionen für die Generierung von PWM-Signalen bereitstellt.

Außerdem mussten die beiden Ultraschallsensoren mit dem Raspberry verbunden werden. Dies geschah über eine kleine selbstverlötete Platine, welche sicherstellen soll, dass die fließenden Spannungen nicht zu hoch sind (die Ultraschallsensoren arbeiten mit 5V und senden auch Signale in dieser Höhe, die GPIO-Pins des Raspberrys dürfen jedoch nicht mehr als 3.3V erhalten).



Schaltplan: Verbindung der Ultraschallsensoren mit dem Raspberry Pi

Um alle Teile des Roboters mit Strom zu versorgen, werden zwei Batterie-Packs verwendet: das eine liefert Strom für den Motorregler und die Motoren, das zweite versorgt den Raspberry Pi, die Ultraschallsensoren sowie das GPS-Modul. Damit der Pi nicht zu viel Spannung erhält, wurde eine Versorgungsplatine (FACHBEGRIFF???) gelötet.

Phase 3: Programmierung

Der Programmcode lässt sich in vier größere Sinnabschnitte gliedern:

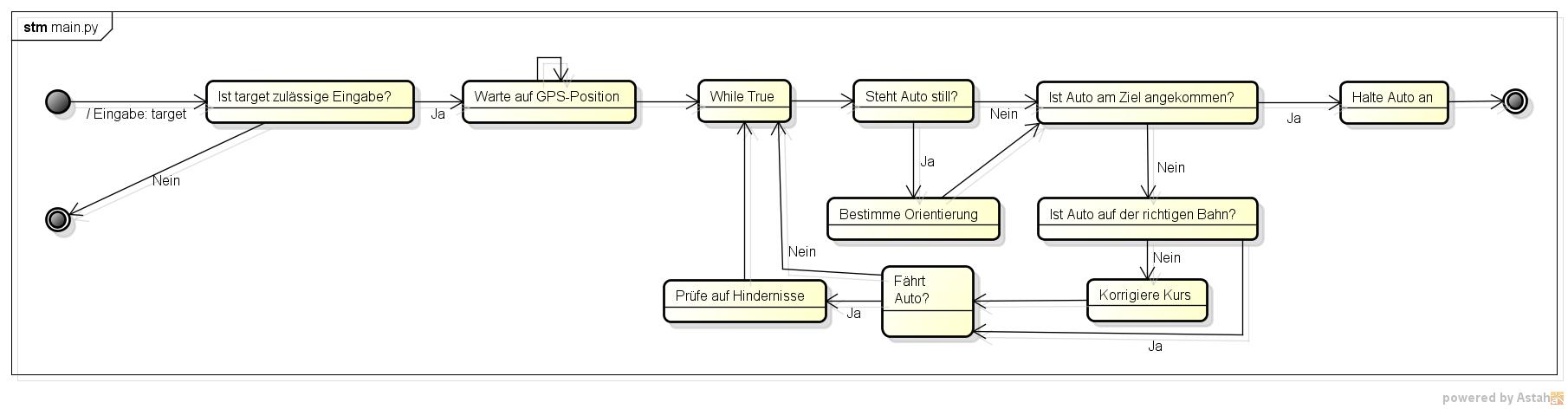
- die Hauptroutine, die das Grundgerüst bildet

- die Navigationsroutine, die GPS-Daten verarbeitet und Routenberechnungen durchführt

- die Hinderniserkennungsroutine, die Gegenständen auf der zu fahrenden Bahn ausweicht

- Hardwarenaher Code, welcher Daten von Sensoren verarbeitet und die Motoren steuert

Im Folgenden wird nun genauer auf diese einzelnen Abschnitte eingegangen.

Hauptroutine:

Überblick über die Funktionsweise der Hauptroutine

Die main-Funktion bedient sich dreier Hilfsfunktionen:

1. approxDistance(current, target)

- approximiert die Distanz zwischen zwei eingegebenen GPS-Koordinaten (aktueller Position und Zielposition)

2. is\_at(current, target)

- prüft ob sich das Auto bis auf einen Schwellenwert genau am Ziel befindet

3. correct\_course(direction, angle, radius, speed=SPEED, watcher=None)

- korrigiert den Kurs durch befahren einer Kreisbahn in der durch direction vorgegebenen Richtung. Die Länge der Kreisbahn ist durch angle und radius vorgegeben. Die Geschwindigkeit ist variabel, es wird jedoch empfohlen so langsam wie möglich zu fahren. Watcher ist ein Objekt der Klasse Watcher, welche für die Hinderniserkennung zuständig ist. Während der Kurskorrektur wird auf Hindernisse geprüft.

Zunächst wird überprüft, ob die über die Kommandozeile beim Starten des Programms eingegebenen Werte gültige Eingaben sind:

if \_\_name\_\_ == '\_\_main\_\_':

try:

target = tuple(float(i) for i in sys.argv[1:3])

except ValueError:

target = None

mainRoutine(target)

und:

def mainRoutine(target):

if target is None or len(target) != 2:

print "no reasonable target given"

return

Sind die Eingaben nicht von der Form "Zahl\_1 Zahl\_2", so beendet das Programm.

Sind die Eingaben richtig, dann werden die nötigen Instanzen unterschiedlicher Klassen erzeugt und initialisiert:

tracker = GPSTracker()

navigator = Navigator(tracker)

navigator.setRadius(RADIUS)

watcher = Watcher()

Die Klasse GPSTracker ermöglicht die Position des Autos sowie dessen Orientierung anhand der letzten Positionen zu bestimmen.

Die Klasse Navigator berechnet den Kurs, der angesteuert werden muss um das Auto zum Ziel zu führen.

Watcher implementiert die Überwachung der Strecke vor dem Auto sowie den Ausweichalgorithmus für etwaige Hindernisse.

Als nächstes muss auf ein GPS-Signal gewartet werden. Sobald dieses vorliegt wird in eine Endlosschleife eingetreten und zunächst überprüft, ob das Auto gerade stillsteht.

Ist dies der Fall, so steht es noch am Anfang seiner Reise. Daher muss zunächst die Orientierung bestimmt werden, damit der Navigator den Kurs berechnen kann. Dazu muss die Liste der letzten Positionen des GPSTrackers gefüllt werden, also fährt das Auto zunächst ca. fünf Meter, füllt dabei über tracker.getPosition() die Liste und hält dann wieder an. Später wurde das GPS-Modul auf einer Stange befestigt, und damit wurde diese Weise der Orientierungsbestimmung überflüssig. In der letzten Version der Software wird die Orientierung über den Kompass abgefragt.

Fährt das Auto gerade, so wird lediglich die Variable driving auf True gesetzt.

while True:

if is\_at(curPos, tracker.getPosition()): # car stands still

print "standing"

start = time.time()

print "driving at", speed

drive(speed) #drive for 5m

while (time.time() - start) < 2: #while driving (ca. 2 s) save positions to tracker

watcher.obstancle()

tracker.getPosition()

stop()

# renavigate

line = None

else:

driving = True

Dann wird überprüft ob das Auto bereits an seinem Ziel angekommen ist. Falls dem so ist, hält es an und das Programm bricht aus der Schleife aus.

Ist es nicht an seinem Ziel, wird überprüft ob sich das Auto auf dem richtigen Kurs befindet und gegebenenfalls auf den korrekten Kurs gebracht.

Die letzte if-Anweisung prüft, ob das Auto gerade fährt - und wenn dem so ist, speichert es seine Position in die Liste der letzten Positionen des GPSTrackers und prüft auf Hindernisse.

Navigationsroutine:

Die Navigationsroutine umschließt drei Dateien:

1. \_\_init\_\_.py

- enthält Klassendefinitionen für GPSPoller und GPSTracker, welche nötig sind um das GPS-Modul anzusteuern.

2. helpers.py

- enthält Hilfsfunktionen für die Berechnungen des Kurses.

3. navigate.py

- enthält die Klassendefinition für Navigator, welche für die Berechnung des Kurses zuständig ist.

Die Klasse Navigator wird mit einem Objekt der Klasse GPSTracker initialisiert.

Die navigate()-Funktion ist das Herzstück der Navigationsroutine.

Nachdem für die aktuelle Position sowie die Zielkoordinate die jeweiligen Längen- und Breitengrade extrahiert wurden, wird die Orientierung des Autos bestimmt.

+++++ Bitte Navigationsroutine weiter ausführen. +++++++

Hinderniserkennungsroutine:

Die Hinderniserkennungsroutine wird durch die Klasse Watcher umgesetzt, welche folgenden Funktionen enthält:

1. alarm(self)

- gibt die zuletzt gemessenen Abstände zu Objekten auf der linken und rechten Seite des Autos zurück.

2. watch(self)

- fragt die Ultraschallsensoren ab und füllt jeweils eine Liste für jeden Sensor mit Messwerten.

3. obstancle(self)

- prüft auf Hindernisse im Bereich von 1.5 Metern vor dem Auto und weicht aus, falls ein Hindernis erkannt wird. Wie stark hierbei gelenkt wird, um auszuweichen ist von der Distanz zum Hindernis abhängig. Das Auto weicht in die Richtung aus, in die eine größere Distanz zum Hindernis gemessen wird. Taucht ein Hindernis unerwarteterweise weniger als einen halben Meter vor dem Auto auf, so wird entweder rückwärts nach links (Hindernis auf der linken Seite; effektiv wird rückwärts nach rechts gefahren) oder rechts (Hindernis auf der rechten Seite; effektiv wird rückwärts nach links gefahren) gesteuert um auszuweichen.

Hardwarenaher Code:

Der hardwarenahe Code umfasst zum einen die Dateien controlServos.py und drive.py, welche für die Ansteuerung der Motoren des Autos zuständig sind und die Fahrtfunktionen enthalten und zum anderen die Datei getDistance.py, welche die Ultraschallsensoren abfragt.

1. controlServos.py

- definiert Variablen für die Länge der zu übermittelnden PWM-Signale für geradeaus lenken/nicht fahren (BASE), rechts lenken (RIGHT), links lenken (LEFT), vorwärts fahren (FORWARD) und rückwärts fahren (BACK). Um Gradierung in Lenkung und Geschwindigkeit zu erlauben, wurden diese Variablen in Abhängigkeit von BASE definiert, sodass man sie mit einem Faktor verwenden kann/muss.

Weiterhin werden die grundlegenden Fahrtfunktionen steerS(deg) und driveS(speed) implementiert, welche jeweils eine Fließkommazahl zwischen -1 (Anschlag rechts bzw. Vollgas rückwärts) und 1 (Anschlag links bzw. Vollgas vorwärts) erwarten.

2. drive.py

- definiert Fahrfunktionen auf einer höheren Ebene. Anstelle von Gradierungen wird drive(speed) die gewünschte Geschwindigkeit in Metern pro Sekunde, steer\_only(radius) der Kurvenradius in Metern und steer\_at(speed, radius) beides übergeben. Die stop()-Funktion hält das Auto ohne Input an, während die stunt()-Funktion ein Easter Egg ist, welche einen kleinen Stunt ausführt.

3. getDistance.py

- setzt benötigte GPIO-Pins des Raspberry Pis so, dass sie für den Input (echo) und Output (trig) für die beiden Ultraschallsensoren verwendet werden können.

Unter Verwendung der Zeit, die die Schallwellen benötigen um nach dem Senden wieder am Sensor anzukommen (timeElapsed) wird die Distanz zum erkannten Objekt in Metern berechnet und zurückgegeben.

Anleitung zur Inbetriebnahme des Roboters/der Software

1. Batteriepacks aufladen (1,5 A; 20mV/C - Ampere-Wert ist variabel).

2. 4000er Batteriepack mit Motorregler verbinden.

3. 8 x 1,2V-Batteriepack mit Stromversorgung auf Plexiglasplatte verbinden.

4. Zugriff auf Raspberry Pi vorbereiten:

a. Bildschirm über HDMI-Kabel mit Raspberry Pi verbinden, Maus und/oder Tastatur über USB anschließen.

b. ALTERNATIV: WLAN-Adapter in USB-Buchse belassen.

5. Stromkabel an Raspberry Pi anschließen (entweder Micro-USB-Kabel der Stromversorgung oder externer Micro-USB-Adapter mit 5V, 1200mA). Warten bis Raspberry Pi gebootet hat (ACT-LED hat aufgehört zu blinken).

6. Falls 4.b. durchgeführt wurde:

Mit einem Rechner Laptop in das nun verfügbare WLAN-Netz "GPS-B" einloggen.

Passwort: "Rasputin".

Linux: Terminal öffnen.

Folgenden Befehl eintippen: "ssh [pi@192.168.7.1](mailto:pi@192.168.7.1)".

Windows: Es kann externe Software wie Putty verwendet werden.

7. Login-Daten für den Raspberry Pi eingeben:

Benutzername: "pi".

Passwort: "raspberry".

ACHTUNG: Raspberry verwendet englisches Tastaturlayout!

8. Verbindung mit dem Internet: LAN-Kabel in zugehörige Buchse am Raspberry Pi stecken. In der Konsole den Befehl

„sudo dhclient eth0“

eintippen, um die LAN-Verbindung freizuschalten.

9. Nun kann man das Labor mit dem Roboter verlassen.

10. Initialisierung des GPS-Sensors über: „sudo ./init\_gpsd.sh“.

11. Initialisierung des Fahrtreglers über:

„sudo python -i“

„from drive import \*“

„drive(0)“

Danach können unterschiedliche Geschwindigkeiten und Steuergrade über die Funktionen drive() und steer() ausprobiert werden. (Zur Sicherheit sollte der Roboter bei diesem Schritt in der Luft gehalten werden.)

12. Mit

„sudo python –i ultrasonicInit.py“

können die Ultraschallsensoren 500 Mal abgefragt werden. Es hat sich gezeigt, dass sie manchmal „warmlaufen“ müssen, daher dieses Skript. Mit dem Aufruf der Funktion init() im Python-Interpreter werden beide Sensoren jeweils ein Mal abgerufen und so kann getestet werden, ob sie erwartungsgemäß funktionieren.

13. Nun kann der Roboter gestartet werden. Um die Hauptroutine zu starten und den Roboter zu vorgegebenen Koordinaten fahren zu lassen und dabei Hindernissen auszuweichen wird im Terminal Folgendes eingegeben:

„sudo python main.py <<Latitude>> <<Longitude>>“

(z.B. „sudo python main.py 49.418056 8.669710“).

Pinbelegung:

|  |  |
| --- | --- |
| **Hardware** | **GPIO-Pin: physisch (BCM)** |
| PWM für Lenkservo | 11 (17) |
| PWM für Fahrtmotor | 15 (22) |
| GPS: RX | 8 (14) |
| GPS: TX | 10 (15) |
| GPS: VCC | 2 |
| GPS: GND | 6 |
| USS links: trig | 12 (18) |
| USS links: echo | 18 (24) |
| USS rechts: trig | 13 (27) |
| USS rechts: echo | 16 (23) |

Aufgetretene Probleme (ggf. mit Lösungsansatz)

ATmega reagierte nicht auf Befehle des Raspberry Pis:

Beim Praxistest des Kommunikationsprotokolls musste festgestellt werden, dass es Komplikationen gab. Obwohl über UART durchaus die richtigen Nachrichten vom Pi gesendet wurden konnte der ATmega diese nicht richtig verarbeiten. Teilweise kamen nur Teile der Nachrichten an, teilweise die komplette Nachricht, allerdings reagierten die Auto-Servos nicht auf diese.

Im Verlauf der Tests, trat ein Fehler bei dem ATmega auf, der den Chip ca. alle 20ms zum Abstürzen brachte. Nach dem Austauschen des ATmega-Chips trat dieses Problem erneut auf, sobald von einem der Rechner im Labor (links neben der Pi-Station) auf den Mikrocontroller geflasht wurde statt von einem Laptop des Teams.

Aufgrund dieser Probleme, die in absehbarer Zeit nicht zu lösen waren, wurde die Entscheidung getroffen, den Roboter ohne ATmega zu betreiben.

Ein weiterer Stolperstein bestand in der richtigen Verwendung des GPS-Moduls bzw. des eingebauten Kompasses. Dieser lieferte unzuverlässige Daten und konnte daher nicht für die Bestimmung der Orientierung verwendet werden.

Gelöst wurde dieses Problem durch das Montieren des GPS-Moduls auf einer Metallstange, welche ihn von dem Rest der Elektronik erhöhte, sodass der elektronische Kompass nicht durch die anderen Komponenten des Roboters gestört wurde.

Ein weiteres Problem des GPS-Moduls ist, dass man es nicht zuverlässig über die externe Stromversorgung versorgen kann wie die Ultraschallsensoren. Daher wird es durch den Pi mit Strom versorgt.

Auch die Ultraschallsensoren zeigten beim Test des Roboters außerhalb des Labors konstant Hindernisse an, welche nicht vorhanden waren. Dies lag daran, dass die Sensoren einen Messwinkel von 15° haben – sowohl horizontal als auch vertikal. Somit wurde der Boden draußen als Hindernis erkannt, da er die Schallwellen zurückwarf. Dieses Problem war im Labor nicht aufgefallen, da das Linoleum die Ultraschallwellen nicht zurückwirft sondern reflektiert und so kein Hindernis wahrgenommen wird.

Um dieses Problem zu lösen wurden die Sensoren durch ein Podest auf dem Roboter erhöht und leicht nach hinten geneigt. Auch die Rückrad-Federung des Autos wurde etwas weicher gestellt, sodass es hinten etwas weiter einsinkt als vorne.

Beim Testen der Navigationsroutine im Zusammenspiel mit der Hinderniserkennungsroutine fuhr der Roboter oft unnötige Kurven und kam so häufig vom Kurs ab. Außerdem wurden Hindernisse manchmal nicht erkannt, weshalb es zu Zusammenstößen kam. Es wurde festgestellt, dass der Roboter nicht schnell genug auf Hindernisse reagieren konnte, da bei der Verarbeitung der Ultraschallsignale eine kleine Verzögerung entsteht.

Daher wurde die Geschwindigkeit des Autos heruntergesetzt und so funktioniert das Zusammenspiel aller Funktionalitäten einwandfrei.

Möchte man mit höheren Geschwindigkeiten fahren, wäre es empfehlenswert bessere Sensoren zu verwenden und/oder den Hinderniserkennungscode (obstancle.py) zu optimieren.

Ausblick

Dieses Projekt bietet viel Potential zur Erweiterung. Denkbar wäre die Hinderniserkennung statt über Ultraschallsensoren über eine Kamera zu implementieren. Dadurch könnten nicht nur genauere Ergebnisse erzielt werden, sie würde auch eine Routenplanung unterstützen, da so der Weg auf eine größere Distanz hin erkundet werden kann.

Routenplanung wäre eine sehr interessante Erweiterung, da die Navigation so auch längere Strecken effizient ermöglichen würde. Zurzeit können zwar auch weiter entfernte Ziele erreicht werden, jedoch muss der Kurs oft korrigiert werden, da nur die Hinderniserkennung dafür sorgt, dass der Roboter auf einem Weg bleibt. Hierfür könnten beispielsweise Daten von OpenStreetMap eingebunden werden.

Die Implementierung einer Geschwindigkeitsmodulation, sodass der Roboter nicht wie aktuell mit nur einer Geschwindigkeit fährt, sondern je nach Route (z.B. wenn eine längere gerade Strecke zu fahren ist) etwas schneller fährt, würde die oben beschriebenen Erweiterungen komplementieren.

Quellcode, frei von erzeugten Binaries

Bilder und ggf. Videos