Programowanie Robota Rovera w Pythonie

Dobrze, sprawdźmy, gdzie teraz jesteś. Masz podstawową wiedzę na temat robotów i (co ważniejsze) głównych komponentów robotów oraz sposobu ich działania. Rozumiesz, że tymi komponentami można sterować za pomocą języka Python i że mogą one współpracować w celu wykonywania zadań robotów. To naprawdę dużo informacji. Następnie pokażemy ci, jak połączyć te rzeczy razem, aby stworzyć bardzo, bardzo prosty „mózg robota", który sprawi, że nasz robot będzie się poruszał samodzielnie. To nie będzie w pełni samojezdny samochód, ale po wykonaniu tej czynności będziesz miał pewne pojęcie o tym, jak te samochody są zaprogramowane

Budowanie prostego interfejsu Pythona wysokiego poziomu

Najpierw stwórzmy krótką funkcję zawijającą w Pythonie, która pozwoli nam budować znacznie bardziej skomplikowane programy, jednocześnie ukrywając złożoność sprzętu robota. Nasz interfejs wysokiego poziomu to plik klasy Pythona o nazwie RobotInterface.py. Długość kodu wykracza poza to, co chcemy pokazać w tej książce, więc opiszmy kilka funkcji, a następnie udokumentujmy resztę.

Funkcja motorForward

Funkcja motorForward jest typowa dla funkcji motorycznych znajdujących się w klasie RobotInterface:

```
def motorForward(self, speed, delay):

motor.motor_left(self.MOTOR_START, self.forward,speed)

motor.motor_right(self.MOTOR_START, self.backward,speed)

time.sleep(delay)

motor.motor_left(self.MOTOR_STOP, self.forward,speed)

motor.motor_right(self.MOTOR_STOP, self.backward,speed)
```

Ta funkcja napędza robota do przodu przez liczbę sekund, które upłynęły w funkcji w argumencie opóźnienia. Oznacza to, że kiedy wywołujesz tę funkcję, musisz użyć rzeczywistej liczby w argumencie opóźnienia. Zasadniczo uruchamia silniki do przodu, czeka z opóźnieniem, a następnie je wyłącza.

Funkcja WheelLeft

```
def wheelsLeft(self):

pwm.set_pwm(self.WHEELS_TURN_SERVO, 0,

calValues.turn_left_max)

time.sleep(0.05)
```

Funkcja wheelsLeft ustawia WHEELS_TURN_SERVO w skrajnej lewej pozycji kół, a następnie opóźnia 50 ms. Dlaczego opóźnienie? Opóźnienia te będą widoczne w pliku klasy RobotInterface. Ma to na celu powstrzymanie wielu poleceń serwa jeden po drugim przed przekroczeniem aktualnej wydajności zasilacza. Opóźniając następne polecenie serwomechanizmu o 50 ms, wysoki prąd przejściowy spowodowany ruchem pierwszego serwomechanizmu ma szansę zaniknąć przed wykonaniem kolejnego polecenia serwomechanizmu.

Funkcja WheelPercent

Ta funkcja pozwala użytkownikowi ustawić serwomechanizm na wartość procentową całkowitego zakresu serwomotoru. Przechodzi od skrajnego lewego (0) do pełnego prawego (100) dla kół; 50 byłoby mniej więcej pośrodku. Może się nieco różnić od środka, jeśli masz asymetryczny zakres ruchu serwa. Jeśli tak, użyj tzw. wheelsMiddle() do centrowania kół. Ten kod oblicza całkowity zakres ruchu serwomotoru, a następnie mnoży o żądany procent. Następnie ustawia serwomotor na żądany zakres:

```
def wheelsPercent(self,percent):

adder = (calValues.turn_left_max –

calValues.turn_right_max)*(percent/100.0)

pwm.set_pwm(self.WHEELS_TURN_SERVO, 0,

int(calValues.turn_right_max adder))

time.sleep(0.05)
```

Wykonywanie pojedynczego ruchu z Pythonem

Przede wszystkim przejdź do strony wsparcia tej książki pod adresem www.dummies.com (patrz Wprowadzenie) i pobierz oprogramowanie dla Księgi 7, rozdział 3. W poniższym kodzie przesuwamy robota na niewielką odległość do przodu za pomocą silnika Do przodu poleceniem, a następnie z powrotem do pierwotnej pozycji za pomocą polecenia motorBackward(). Mamy nadzieję, że zaczynasz dostrzegać magię tego podejścia. Kod „Pojedynczy ruch”:

```
#!/usr/bin/python3

# Robot Interface Test

import RobotInterface

import time

RI = RobotInterface.RobotInterface()

print ("Short Move Test")

RI.wheelsMiddle()

RI.motorForward(100,1.0)

time.sleep(1.0)

RI.motorBackward(100,1.0)
```

Najpierw importujemy bibliotekę klas RobotInterface, a także bibliotekę czasu (dla funkcji sleep()). Zwróć uwagę na prostotę tego. Złożoność leżąca u podstaw bibliotek interfejsu robota jest ukryta w tej klasie:

```
import RobotInterface

import time
```

Następnie inicjujemy moduł RobotInterface i przypisujemy moduł do zmiennej RI:

```
RI = RobotInterface.RobotInterface()

print ("Short Move Test")
```

Centrujemy koła za pomocą polecenia funkcji wheelsMiddle():

RI.wheelsMiddle()

Teraz nadchodzi dobra część. Prowadzimy robota do przodu przez jedną sekundę, zatrzymujemy się na sekundę, a następnie cofamy go na jedną sekundę do pierwotnej pozycji robota:

RI.motorForward(100,1.0)

time.sleep(1.0)

RI.motorBackward(100,1.0)

Całkiem proste, prawda? Wywołaj plik singleMove.py. Oto film pokazujący, co powinieneś zobaczyć, gdy uruchomisz ten kod na swoim robocie w oknie terminala: https://youtu.be/UT0PG7z2ccE. Praca nie jest skończona do kiedy? O tak, kiedy dokumentacja jest gotowa. O™ w celu udokumentowania funkcji klasy RobotInterface.

Funkcje klasy RobotInterface

W tej sekcji dokumentujemy funkcje klasy RobotInterface. Pokazujemy program Robot Interface Test, a potem czas na wyścigi w budowaniu oprogramowania dla robotów! Klasa RobotInterface wywodzi się zarówno z oryginalnego oprogramowania autora, jak i z wewnętrznych sterowników oprogramowania PiCar-B Adeept.

Funkcje przednich diod LED

Następujące funkcje kontrolują dwie diody LED z przodu robota.

set_Front_LED_On()

Ta funkcja włącza przednią diodę LED:

set_Front_LED_On(colorLED)

Pamiętaj, że te dwie przednie diody LED są trójkolorowe z czerwonymi, zielonymi i niebieskimi diodami LED, z których każda jest indywidualnie sterowana. Parametr colorLED kontroluje stronę robota, która ma się włączyć, oraz kolor, który ma się włączyć. Kolor i stronę wybierasz za pomocą następujących stałych z klasy RobotInterface:

RobotInterface.left_R

RobotInterface.left_G

RobotInterface.left_B

RobotInterface.right_R

RobotInterface.right_G

RobotInterface.right_B

Na przykład RobotInterface.left_R włącza czerwoną diodę LED po lewej stronie robota. Lewy robot odnosi się do lewej strony robota widzianej z tyłu robota. Możesz wykonać wiele wywołań tego programu, aby włączyć wszystkie trzy diody LED. Włączenie już włączonej diody nic nie szkodzi i jest ignorowane. Można napisać bardziej wyrafinowany sterownik, który steruje diodami LED GPIO z PWM (modulacja szerokości impulsu), umożliwiając jeszcze większe mieszanie kolorów. Zwróć uwagę, że jeśli

nie używasz sprzętowych pinów PWM na Raspberry Pi, podczas korzystania z tej techniki zobaczysz migotanie diod LED ze względu na wielozadaniowy system operacyjny Raspberry Pi. Możesz jednak napisać sterowniki dla karty sterownika serwo PCA9685 (na PiCar-B), która obecnie steruje serwosilnikami, aby rozwiązać problem migotania.

set_Front_LED_Of()

Ta funkcja wyłącza przednią diodę LED:

set_Front_LED_Off(colorLED)

Pamiętaj, że te dwie przednie diody LED są trójkolorowe z czerwonymi, zielonymi i niebieskimi diodami LED, z których każda jest indywidualnie sterowana. Parametr colorLED kontroluje stronę robota, która ma się włączyć, oraz kolor, który ma się włączyć. Kolor i stronę wybierasz za pomocą następujących stałych z klasy RobotInterface:

RobotInterface.left_R

RobotInterface.left_G

RobotInterface.left_B

RobotInterface.right_R

RobotInterface.right_G

RobotInterface.right_B

Na przykład RobotInterface.left_R włącza czerwoną diodę LED po lewej stronie robota. Lewy robot odnosi się do lewej strony robota widzianej z tyłu robota. Możesz wykonać wiele wywołań tego programu, aby włączyć wszystkie trzy diody LED. Włączenie już włączonej diody nic nie szkodzi i jest ignorowane.

Funkcje paska pikseli

Na robocie znajduje się 12 diod LED połączonych razem jako pojedynczy pasek 12 diod LED. Te diody LED RGB nazywane są pikselami i są kontrolowane przez pojedynczą linię szeregową, która biegnie przez wszystkie 12 diod LED. Są kontrolowane przez dość wyrafinowaną i drażliwą sekwencję szeregową precyzyjnie zsynchronizowanych impulsów z Raspberry Pi. Raspberry Pi (ponownie ze względu na system operacyjny) nie może generować tych impulsów wystarczająco dokładnie za pomocą sygnałów Pythona i GPIO. Dlatego złożony sterownik wykorzystujący interfejs DMA (bezpośredni dostęp do pamięci) na Raspberry Pi został stworzony przez „jgarff" (rpi_ws281x — bardzo sprytne kodowanie) i używamy tego sterownika do generowania tych sygnaów. Nasze oprogramowanie RobotInterface ukrywa całą tę złożoność przed użytkownikiem.

rainbowCycle()

To wywołanie rozpoczyna cykl tęczy, który wykorzystuje wszystkie 12 diod LED Pixel i przebiega przez wiele kolorów:

rainbowCycle(wait_ms = 20, iterations = 3)

Parametr wait_ms ustawia opóźnienie (w milisekundach) pomiędzy każdą zmianą koloru. Domyślnie jest to 20 ms. Interations ustawia liczbę pełnych cykli kolorów do wykonania przed powrotem, a domyślną wartością jest 3. colorWipe() Ta funkcja ustawia wszystkie 12 diod Pixel na ten sam kolor:

colorWipe(color)

Na przykład colorWipe(color(0,0,0)) ustawia wszystkie piksele na Off. Jest to wygodny sposób ustawienia wszystkich 12 pikseli na ten sam kolor. Parametr color określa kolor, który ma być użyty przy użyciu funkcji color(). (Zobacz funkcję color()).

teatrChaseRainbow()

Ta funkcja uruchamia 40-sekundowy wzór ścigania diod LED na wszystkich 12 diodach LED:

theaterChaseRainbow(wait_ms = 50)

Parametr wait_ms ustawia opóźnienie między każdym ruchem w milisekundach. Domyślnie jest to 50 milisekund.

setPixelColor()

Ta funkcja ustawia pojedynczy piksel (ponumerowany od 0 do 11) na określony kolor. Jasność wpływa na wszystkie piksele i wykorzystuje ostatnią ustawioną wartość jasności:

setPixelColor(pixel, color, brightness)

Parametr pixel ustawia konkretny piksel do ustawienia. Piksele są ponumerowane od 0 do 11. Parametr color określa kolor, który ma być użyty za pomocą funkcji color(). Parametr jasności ustawia jasność (0–255) dla całego ciągu pikseli.

Color()

Ta funkcja jest funkcją pomocniczą, która konwertuje wartości R, G i B na pojedynczą 24-bitową liczbę całkowitą używaną przez wewnętrzny sterownik Pixela:

Color(red, green, blue, white = 0)

Parametry czerwony, zielony i niebieski są liczbami całkowitymi i mieszczą się w zakresie od 0 (wyłączony) do 255 (w pełni włączony). Parametr white=0 dotyczy diod RGBW Pixel. Piksele na robocie to diody LED RGB.

wszystkieLEDSOf()

Ta funkcja wyłącza wszystkie diody LED robota, zarówno dwie przednie diody LED, jak i ciąg 12 pikseli LED:

allLEDSOff()

Funkcja ultradźwiękowego czujnika odległości.Ultradźwiękowy czujnik odległości działa poprzez wysyłanie impulsu dźwięku o wysokiej częstotliwości, a następnie odliczanie czasu, zanim odbije się on z powrotem do odbiornika. Ponieważ znamy prędkość dźwięku, możemy obliczyć odległość przed czujnikiem. Nie jest to idealna metoda (wolelibyśmy użyć lasera!), Ale jest całkiem dobra i stanowi dobry czujnik odległości początkowej.

fetchUltraDistance()

Ta funkcja wykonuje natychmiastowy pomiar z ultradźwiękowego czujnika odległości w głowicy robota i zwraca odległość w centymetrach (cm):

fetchUltraDistance()

Główne funkcje motoryczne

Główny silnik naszego robota napędza tylne koła i wprawia robota w ruch. Funkcje silnika służą do określenia, jak szybko i jak długo ma pracować główny silnik.

motorForward()

Ta funkcja napędza robota do przodu z prędkością przez liczbę sekund opóźnienia przed wyłączeniem silników:

motorForward(prędkość, opóźnienie)

Parametr prędkość określa cykl pracy pinu PWM GPIO sterującego interfejsem silnika. Zmienia się od 0 (o) do 100 (szybko).

Opóźnienie parametru mówi kierowcy, jak długo ma pracować silnik w sekundach.

motorBackward()

Ta funkcja powoduje cofanie robota do tyłu z prędkością przez liczbę sekund opóźnienia przed wyłączeniem silników:

motorBackward(prędkość, opóźnienie)

Parametr prędkość określa cykl pracy pinu PWM GPIO sterującego interfejsem silnika. Zmienia się od 0 (o) do 100 (szybko). Opóźnienie parametru mówi kierowcy, jak długo ma pracować silnik w sekundach.

stopMotor()

Ta funkcja natychmiast zatrzymuje silnik główny:

stopMotor()

Jest to naprawdę przydatne tylko wtedy, gdy na przykład napędzałeś silnik w innym wątku. Możesz myśleć o wątku jako o innym programie działającym w tym samym czasie, co program główny. Jest to wyrafinowana technika programowania, która ma ogromne zalety w pisaniu kodu robota.

Funkcje serwa

Ta grupa funkcji kontroluje trzy serwomechanizmy robota: obracanie głową, przechylanie głowy i przednie koła.

headTurnLeft()

Ta funkcja obraca głowicę robota maksymalnie w lewo:

headTurnLeft()

„Do końca w lewo” jest określone w pliku calValues.py.

headTurnRight()

Ta funkcja obraca głowicę robota maksymalnie w prawo:

headTurnRight()

„Do końca w prawo” jest określone w pliku calValues.py.

headTurnMiddle()

Ta funkcja obraca głowicę robota do przodu:

headTurnMiddle()

„Środek” jest określony w pliku calValues.py.

HeadTurnPercent()

Ta funkcja obraca głowę od 0 (całkowicie w lewo) do 100 (całkowicie w prawo):

headTurnPercent(procent)

Jest to przydatne do dokładniejszego celowania głową. Ponownie, „całkowicie w lewo” i „całkowicie w prawo” są zdefiniowane w pliku calValues.py. Informacje na temat wartości kalibracji i sposobu ich ustawiania za pomocą programu calibrateServos.py znajdują się w rozdziale 2 tego minibooka. Parametr procent ma wartości 0–100 i reprezentuje procent liniowy od lewej do prawej. Zauważ, że wartość 50 może nie być całkiem pośrodku, ponieważ twoje serwomechanizmy mogą nie być ustawione dokładnie w środku swojego zakresu.

headTiltDown()

Ta funkcja całkowicie przechyla głowicę robota w dół:

headTiltDown()

„Do końca” jest określone w pliku calValues.py.

headTiltUp()

Ta funkcja całkowicie przechyla głowę robota:

headTiltUp()

„Całkowicie w górę” jest określone w pliku calValues.py.

headTiltMiddle()

Ta funkcja przechyla głowę robota do przodu:

headTiltMiddle()

„Środek” jest określony w pliku calValues.py.

headTiltPercent()

Ta funkcja obraca głowicę od 0 (całkowicie w dół) do 100 (całkowicie w górę):

headTiltPercent(procent)

Jest to przydatne do dokładniejszego celowania głową. Ponownie, „całkowicie w dół” i „całkowicie w górę”” są zdefiniowane w pliku calValues.py. Parametr procent ma wartości od 0 do 100 i reprezentuje procent liniowy od dołu do góry. Zwróć uwagę, że wartość 50 może nie być całkiem pośrodku, ponieważ serwa mogą nie być ustawione dokładnie w środku ich zakresu ustawionego w procesie kalibracji serwomechanizmu oraz ze względu na fizyczną budowę robota.

wheelsLeft()

Ta funkcja obraca przednie koła robota maksymalnie w lewo:

wheelsLeft()

„Do końca w lewo” jest określone w pliku calValues.py.

wheelsRight()

Ta funkcja obraca przednie koła robota maksymalnie w prawo:

wheelsRight()

„Do końca w prawo” jest określone w pliku calValues.py.

wheelsMiddle()

Ta funkcja obraca przednie koła robota do środka:

wheelsMiddle()

„Średnia” jest zdefiniowana w pliku calValues.py.

wheelsPercent()

Ta funkcja obraca głowę od 0 (całkowicie w lewo) do 100 (całkowicie w prawo):

wheelsPercent(percent)

Jest to przydatne do dokładniejszego ustawienia kierunku przednich kół robota. Ponownie, „całkowicie w lewo i „całkowicie w prawo”” są zdefiniowane w pliku calValues.py. Parametr procent ma wartości 0-100 i reprezentuje procent liniowy od dołu do góry. Zwróć uwagę, że wartość 50 może nie być całkiem pośrodku ze względu na to, że serwomechanizmy mogą nie być ustawione dokładnie w środku zakresu ustawionego w procesie kalibracji serwomechanizmu oraz z powodu fizycznej budowy robota.

Ogólna funkcja serwomechanizmu

Dołączyliśmy ogólne funkcje do sterowania wszystkimi serwami jednocześnie. Wywołanie tej funkcji przesuwa wszystkie serwomechanizmy do pozycji środkowej.

centerAllServos()

Ta funkcja umieszcza wszystkie serwomechanizmy robota w środku ich zakresów, jak określono w pliku calValues.py:

centerAllServos()

Test interfejsu robota Pythona.

Teraz, gdy mamy już określone nasze API robota (interfejs programowania aplikacji), uruchommy test systemu przy użyciu klasy RobotInterface Pythona. Ten program jest przydatny z dwóch powodów. Najpierw testuje wszystkie nasze funkcje w klasie RobotInterface. Po drugie, pokazuje, jak używać każdej z funkcji w programie Pythona. Kod dla RITest.py:

```
#!/usr/bin/python3

# Robot Interface Test

import RobotInterface
```

```
import time
RI = RobotInterface.RobotInterface()
print ("Robot Interface Test")
print ("LED tests")
RI.set_Front_LED_On(RI.left_R)
time.sleep(0.1)
RI.set_Front_LED_On(RI.left_G)
time.sleep(0.1)
RI.set_Front_LED_On(RI.left_B)
time.sleep(1.0)
RI.set_Front_LED_On(RI.right_R)
time.sleep(0.1)
RI.set_Front_LED_On(RI.right_G)
time.sleep(0.1)
RI.set_Front_LED_On(RI.right_B)
time.sleep(1.0)
RI.set_Front_LED_Off(RI.left_R)
time.sleep(0.1)
RI.set_Front_LED_Off(RI.left_G)
time.sleep(0.1)
RI.set_Front_LED_Off(RI.left_B)
time.sleep(1.0)
RI.set_Front_LED_Off(RI.right_R)
time.sleep(0.1)
RI.set_Front_LED_Off(RI.right_G)
time.sleep(0.1)
RI.set_Front_LED_Off(RI.right_B)
time.sleep(1.0)
RI.rainbowCycle(20, 1)
time.sleep(0.5)
# Runs for 40 seconds
```

```
#RI.theaterChaseRainbow(50)
#time.sleep(0.5)
print ("RI.Color(0,0,0)=", RI.Color(0,0,0))
RI.colorWipe(RI.Color(0,0,0))
time.sleep(1.0)
for pixel in range (0,12):
RI.setPixelColor(pixel,RI.Color(100,200,50),50)
time.sleep(0.5)
print ("Servo Tests")
RI.headTurnLeft()
time.sleep(1.0)
RI.headTurnRight()
time.sleep(1.0)
RI.headTurnMiddle()
time.sleep(1.0)
RI.headTiltDown()
time.sleep(1.0)
RI.headTiltUp()
time.sleep(1.0)
RI.headTiltMiddle()
time.sleep(1.0)
RI.wheelsLeft()
time.sleep(1.0)
RI.wheelsRight()
time.sleep(1.0)
RI.wheelsMiddle()
time.sleep(1.0)
print("servo scan tests")
for percent in range (0,100):
RI.headTurnPercent(percent)
for percent in range (0,100):
```

```
RI.headTiltPercent(percent)

for percent in range (0,100):

RI.wheelsPercent(percent)

print("motor test")

RI.motorForward(100,1.0)

time.sleep(1.0)

RI.motorBackward(100,1.0)

print("ultrasonic test")

print ("distance in cm=", RI.fetchUltraDistance())

print("general function test")

RI.allLEDSOff()

RI.centerAllServos()
```

Uwaga: Skomentowaliśmy kod testowy RI.theatreChaseRainbow(), ponieważ działa on przez 40 sekund

ROS: SYSTEM OPERACYJNY ROBOTA

Napisaliśmy dość prostą klasę interfejsu dla robota PiCar-B. Dzięki temu możemy sterować robotem z poziomu programu w Pythonie. Gdybyśmy mieli więcej miejsca w tej książce (właściwie można by napisać całą inną książkę o wykorzystaniu ROS dla robota takiego jak nasz), podłączylibyśmy naszego robota do ROS (Robot Operating System). ROS to system specjalnie zaprojektowany do sterowania robotami w systemie rozproszonym. Mimo że nazywa się to Robot Operating System, tak naprawdę nie jest systemem operacyjnym. ROS to tak zwane oprogramowanie pośredniczące. Oprogramowanie pośrednie to oprogramowanie zaprojektowane do zarządzania złożonością pisania oprogramowania w złożonym i heterogenicznym środowisku (co oznacza wiele różnych typów robotów i czujników). ROS pozwala nam traktować bardzo różne roboty w bardzo podobny sposób. ROS działa w oparciu o tak zwany system publikowania-subskrybowania. Działa jak gazeta. Gazeta publikuje historie, ale tylko ludzie, którzy ją prenumerują, widzą te historie. Możesz mieć subskrybenta, który chce tylko subskrypcji komiksów. Albo na pierwszą stronę. ROS tak działa. Robot taki jak nasz może publikować aktualną wartość czujnika ultradźwiękowego lub aktualny obraz z kamery (lub nawet strumień wideo), a inne komputery lub roboty w sieci mogą subskrybować strumień wideo i widzieć, co widzi twój robot. A twój robot może subskrybować inne czujniki (takie jak czujnik temperatury umieszczony na środku pokoju) lub nawet patrzeć na to, co widzą inne roboty. Siła tej techniki polega na tym, że teraz możesz uczynić swojego robota częścią ekosystemu składającego się z komputerów, czujników, a nawet ludzi korzystających z twoich danych i przekazujących informacje do twojego robota. Moglibyśmy zbudować interfejs ROS na naszym robocie, a następnie sterować nim zdalnie i przesyłać dane z czujników do innych komputerów. Pod wieloma względami ROS naprawdę rządzi. Dowiedz się więcej o ROS na http://www.ros.org/.

Uruchom program, wpisując sudo python3 RITest.py w oknie terminala. Pamiętaj, że musisz użyć sudo, ponieważ diody LED pikseli wymagają uprawnień administratora (przyznanych przez sudo), aby poprawnie działać.

Robot Interface Test

LED tests

RI.Color(0,0,0)= 0

Servo Tests

servo scan tests

motor test

ultrasonic test

distance in cm= 16.87312126159668

general function test

Koordynacja ruchów motorycznych z czujnikami

Zdolność do modyfikowania i koordynowania ruchów silnika z ruchami czujników jest kluczem do ruchu w środowisku. Czujniki przekazują informacje, na podstawie których należy działać, a także informacje zwrotne z naszych ruchów. Pomyśl o akcie łapania piłki baseballowej rękawiczką. Twoje czujniki? Oczy i zmysł dotyku. Twoje oczy widzą piłkę, a następnie poruszają ręką i ramieniem, aby przechwycić piłkę. To koordynowanie ruchu za pomocą czujnika. Informacje zwrotne? Wiedza o tym, że złapałeś piłkę w rękawicy, po odczuciu, jak uderza ona w dłonie w rękawiczkach. Oczywiście aktualizujesz również swój wewnętrzny system uczenia się, aby lepiej łapać piłkę. PiCar-B ma dwa czujniki, które odczytują informacje ze świata zewnętrznego. Ultradźwiękowy czujnik może wykryć, co znajduje się przed robotem, podczas gdy kamera może fotografować świat, a następnie robot może analizować to, co widzi. Pierwszą rzeczą do zapamiętania jest jednak to, że widzenie robota jest trudne. Bardzo trudny. W następnym rozdziale tej książki poruszymy temat wykorzystania obrazów z kamery do analizy. Rozdział 4 mówi o wykorzystaniu sztucznej inteligencji w robotach, a my zbudujemy przykład, jak to zrobić za pomocą uczenia maszynowego. W naszym przykładzie skupimy się na prostszym czujniku, ultradźwiękowym czujniku odległości. Oto przykład kodu, który przesunie robota do przodu lub do tyłu w zależności od odległości od obiektu przed robotem. Oto kod Pythona dla simpleFeedback.py: Koordynacja ruchów motorycznych z czujnikami Możliwość modyfikowania i koordynowania ruchów motorycznych z ruchami czujników jest kluczem do poruszania się w środowisku. Czujniki przekazują informacje, na podstawie których należy działać, a także informacje zwrotne z naszych ruchów. Pomyśl o akcie łapania piłki baseballowej rękawiczką. Twoje czujniki? Oczy i zmysł dotyku. Twoje oczy widzą piłkę, a następnie poruszają ręką i ramieniem, aby przechwycić piłkę. To koordynowanie ruchu za pomocą czujnika. Informacje zwrotne? Wiedza o tym, że złapałeś piłkę w rękawicy, po odczuciu, jak uderza ona w dłonie w rękawiczkach. Oczywiście aktualizujesz również swój wewnętrzny system uczenia się, aby lepiej łapać piłkę. PiCar-B ma dwa czujniki, które odczytują informacje ze świata zewnętrznego. Czujnik ultradźwiękowy może wykrywać to, co znajduje się przed robotem, podczas gdy kamera może fotografować świat, a następnie robot może analizować to, co widzi. Pierwszą rzeczą do zapamiętania jest jednak to, że widzenie robota jest trudne. Bardzo trudny. W następnym rozdziale tej książki poruszymy temat wykorzystania obrazów z kamery do analizy. Rozdział 4 mówi o wykorzystaniu sztucznej inteligencji w robotach, a my zbudujemy przykład, jak to zrobić za pomocą uczenia maszynowego. W naszym przykładzie skupimy się na prostszym czujniku, ultradźwiękowym czujniku odległości. Oto przykład kodu, który przesunie robota do przodu lub do tyłu w zależności od odległości od obiektu przed robotem. Oto kod Pythona dla simpleFeedback.py:

#!/usr/bin/python3

```
# Robot Interface Test
import RobotInterface
import time
DEBUG = True
RI = RobotInterface.RobotInterface()
print ("Simple Feedback Test")
RI.centerAllServos()
RI.allLEDSOff()
# Ignore distances greater than one meter
DISTANCE_TO_IGNORE = 1000.0
# Close to 10cm with short moves
DISTANCE_TO_MOVE_TO = 10.0
# How many times before the robot gives up
REPEAT_MOVE = 10
def bothFrontLEDSOn(color):
RI.allLEDSOff()
if (color == "RED"):
RI.set_Front_LED_On(RI.right_R)
RI.set_Front_LED_On(RI.left_R)
return
if (color == "GREEN"):
RI.set_Front_LED_On(RI.right_G)
RI.set_Front_LED_On(RI.left_G)
return
if (color == "BLUE"):
RI.set_Front_LED_On(RI.right_B)
RI.set_Front_LED_On(RI.left_B)
return
try:
Quit = False
moveCount = 0
```

```
bothFrontLEDSOn("BLUE")
while (Quit == False):
current_distance = RI.fetchUltraDistance()
if (current_distance >= DISTANCE_TO_IGNORE):
bothFrontLEDSOn("BLUE")
if (DEBUG):
print("distance too far ={:6.2f}cm"
.format(current_distance))
else:
if (current_distance <= 10.0):
# reset moveCount
# the Robot is close enough
bothFrontLEDSOn("GREEN")
moveCount = 0
if (DEBUG):
print("distance close enough ={:6.2f}cm"
.format(current_distance))
time.sleep(5.0)
# back up and do it again
RI.motorBackward(100,1.0)
else:
if (DEBUG):
print("moving forward ={:6.2f}cm"
.format(current_distance))
# Short step forward
bothFrontLEDSOn("RED")
RI.motorForward(90,0.50)
moveCount = moveCount 1
# Now check for stopping our program
time.sleep(1.0)
if (moveCount > REPEAT_MOVE):
```

```
Quit = True

except KeyboardInterrupt:

print("program interrupted")

print ("program finished")
```

To doskonały przykład wykorzystania sprzężenia zwrotnego w robotyce. Robot najpierw sprawdza, czy znajduje się mniej niż jeden metr (1000 cm) od ściany. Jeśli tak, powoli zaczyna zbliżać się do ściany małymi krokami. Gdy znajdzie się bliżej niż 10 cm od ściany, zatrzymuje się, następnie odczekuje pięć sekund i cofa się, aby zrobić to ponownie. Poddaje się również, jeśli dotarcie do ściany zajmuje więcej niż 10 ruchów, jeśli w jakiś sposób odsunęliśmy się od ściany dalej niż 1000 cm lub jeśli użytkownik wcisnął Ctrl-C aby przerwać program. Zwróć uwagę, jak używamy diod LED, aby przekazać informacje zwrotne otaczającym ludziom, co robi robot. Ten rodzaj wizualnej informacji zwrotnej jest ważnym elementem zwiększania wydajności, zrozumiałości i bezpieczeństwa interakcji człowiek-robot. Główna struktura programu jest zawarta w pętli while Pythona. Dopóki nie przerwiemy programu (lub nie zostanie spełnione jedno z pozostałych kryteriów wyjścia), nasz mały robot będzie pracował do wyczerpania baterii. Skopiuj kod do simpleFeedback.py i wypróbuj go, wykonując sudo python3 simpleFeedback.py. Oto wydrukowane wyniki:

```
Simple Feedback Test

moving forward = 55.67cm

moving forward = 44.48cm

moving forward = 34.22cm

moving forward = 26.50cm

moving forward = 17.53cm

distance close enough = 9.67cm

moving forward = 66.64cm

moving forward = 54.25cm

moving forward = 43.55cm

moving forward = 36.27cm

moving forward = 28.44cm

moving forward = 21.08cm

moving forward = 13.55cm

distance close enough = 6.30cm

moving forward = 64.51cm

moving forward = 52.89cm

moving forward = 43.75cm

moving forward = 33.95cm
```

moving forward = 26.79cm

^Cprogram interrupted

program finished

Film zwrotny można zobaczyć tutaj: https://youtu.be/mzZIMxch5k4. Zagraj z tym kodem. Wypróbuj różne rzeczy i różne stałe, aby uzyskać różne wyniki.

Tworzenie mózgu Pythona dla naszego robota

Teraz stworzymy prosty samojezdny samochód. W pewnym sensie zamierzamy zastosować powyższe wyniki do stworzenia autonomicznego pojazdu, który nie jest zbyt inteligentny, ale ilustruje wykorzystanie informacji zwrotnych w podejmowaniu decyzji. Mózg Pythona, który piszemy, jest niczym więcej niż kombinacją naszego kodu do wykrywania ściany i do generowania przypadkowego spaceru na podstawie informacji. Po uruchomieniu kodu przez chwilę zobaczyliśmy, gdzie robot utknął, i dodaliśmy kod wykrywający powrót z zablokowanych pozycji.

Uwaga: Upewnij się, że masz w pełni naładowane baterie, aby uruchomić ten kod. Kiedy baterie nieco się wyczerpią, prędkość twojego silnika dramatycznie spada. Jak to naprawić? Używaj większych baterii.

Kod „mózgu robota”:

```
#!/usr/bin/python3

# Robot Brsin

import RobotInterface

import time

from random import randint

DEBUG = True

RI = RobotInterface.RobotInterface()

print ("Simple Robot Brain")

RI.centerAllServos()

RI.allLEDSOff()

# Close to 20cm

CLOSE_DISTANCE = 20.0

# How many times before the robot gives up

REPEAT_TURN = 10

def bothFrontLEDSOn(color):

RI.allLEDSOff()

if (color == "RED"):

RI.set_Front_LED_On(RI.right_R)
```

```
RI.set_Front_LED_On(RI.left_R)
return
if (color == "GREEN"):
RI.set_Front_LED_On(RI.right_G)
RI.set_Front_LED_On(RI.left_G)
return
if (color == "BLUE"):
RI.set_Front_LED_On(RI.right_B)
RI.set_Front_LED_On(RI.left_B)
return
STUCKBAND = 2.0
# check for stuck car by distance not changing
def checkForStuckCar(cd,p1,p2):
if (abs(p1-cd) < STUCKBAND):
if (abs(p2-cd) < STUCKBAND):
return True
return False
try:
Quit = False
turnCount = 0
bothFrontLEDSOn("BLUE")
previous2distance = 0
previous1distance = 0
while (Quit == False):
current_distance = RI.fetchUltraDistance()
if (current_distance >= CLOSE_DISTANCE ):
bothFrontLEDSOn("BLUE")
if (DEBUG):
print("Continue straight ={:6.2f}cm"
.format(current_distance))
if (current_distance > 300):
```

```
# verify distance
current_distance = RI.fetchUltraDistance()
if (current_distance > 300):
# move faster
RI.motorForward(90,1.0)
else:
RI.motorForward(90,0.50)
turnCount = 0
else:
if (DEBUG):
print("distance close enough so turn ={:6.2f}cm"
.format(current_distance))
bothFrontLEDSOn("RED")
# now determine which way to turn
# turn = 0 turn left
# turn = 1 turn right
turn = randint(0,1)
if (turn == 0): # turn left
# we turn the wheels right since
# we are backing up
RI.wheelsRight()
else:
616 BOOK 7 Building Robots with Python
# turn right
# we turn the wheels left since
# we are backing up
RI.wheelsLeft()
time.sleep(0.5)
RI.motorBackward(100,1.00)
time.sleep(0.5)
RI.wheelsMiddle()
```

```
turnCount = turnCount 1
print("Turn Count =", turnCount)
# check for stuck car
if (checkForStuckCar(current_distance,
previous1distance, previous2distance)):
# we are stuck. Try back up and try Random turn
bothFrontLEDSOn("RED")
if (DEBUG):
print("Stuck - Recovering ={:6.2f}cm"
.format(current_distance))
RI.wheelsMiddle()
RI.motorBackward(100,1.00)
# now determine which way to turn
# turn = 0 turn left
# turn = 1 turn right
turn = randint(0,1)
if (turn == 0): # turn left
# we turn the wheels right since
# we are backing up
RI.wheelsRight()
else:
# turn right
# we turn the wheels left since
# we are backing up
RI.wheelsLeft()
time.sleep(0.5)
RI.motorBackward(100,2.00)
time.sleep(0.5)
RI.wheelsMiddle()
# load state for distances
previous2distance = previous1distance
```

```
previous1distance = current_distance
```

CHAPTER 3 Programming Your Robot Rover in Python 617

```
# Now check for stopping our program
time.sleep(0.1)
if (turnCount > REPEAT_TURN-1):
bothFrontLEDSOn("RED")
if (DEBUG):
print("too many turns in a row")
Quit = True
except KeyboardInterrupt:
print("program interrupted")
print ("program finished")
```

Wydaje się, że jest to znacznie bardziej złożony program niż nasz program czujnika ultradźwiękowego przedstawiony wcześniej w tym rozdziale, ale w rzeczywistości tak nie jest.

Wzięliśmy tę samą strukturę programu (pętla while) i dodaliśmy kilka funkcji. Najpierw dodaliśmy klauzulę rozpędzania samochodu, gdy byliśmy daleko od przeszkody (powyżej 300 cm):

```
if (current_distance >= CLOSE_DISTANCE ):
bothFrontLEDSOn("BLUE")
if (DEBUG):
print("Continue straight ={:6.2f}cm"
.format(current_distance))
if (current_distance > 300):
# verify distance
current_distance = RI.fetchUltraDistance()
if (current_distance > 300):
# move faster
RI.motorForward(90,1.0)
else:
RI.motorForward(90,0.50)
turnCount = 0
```

Kontynuowaliśmy poruszanie się małymi skokami, gdy robot zbliżał się do ściany. Kiedy robot zbliży się do ściany na odległość około 10 cm, postanawia skręcić przednimi kołami w przypadkowym kierunku i cofa się, aby wypróbować nowy kierunek:

```
if (DEBUG):

print("distance close enough so turn ={:6.2f}cm"

.format(current_distance))

bothFrontLEDSOn("RED")

# now determine which way to turn

# turn = 0 turn left

# turn = 1 turn right

turn = randint(0,1)

if (turn == 0): # turn left

# we turn the wheels right since

# we are backing up

RI.wheelsRight()

else:

# turn right

# we turn the wheels left since

# we are backing up

RI.wheelsLeft()

time.sleep(0.5)

RI.motorBackward(100,1.00)

time.sleep(0.5)

RI.wheelsMiddle()

turnCount = turnCount 1

print("Turn Count =", turnCount)
```

Uruchomiliśmy robota przez dość długi czas, stosując właśnie tę logikę, i zobaczylibyśmy, że utknąłby, gdyby część robota była zablokowana, ale czujnik ultradźwiękowy nadal wykrywał odległość większą niż 10 cm. Aby temu zaradzić, dodaliśmy bieżący zapis ostatnich dwóch ultradźwiękowych odczytów odległości, a jeśli masz trzy odczyty /- 2,0 cm, robot uzna, że utknął i cofnie się, obróci losowo i ponownie zacznie wędrować. Pracował jak mistrz:

```
if (checkForStuckCar(current_distance,

previous1distance, previous2distance)):
```

```
# we are stuck. Try back up and try Random turn
bothFrontLEDSOn("RED")
if (DEBUG):
print("Stuck - Recovering ={:6.2f}cm"
.format(current_distance))
RI.wheelsMiddle()
RI.motorBackward(100,1.00)
# now determine which way to turn
# turn = 0 turn left
# turn = 1 turn right
turn = randint(0,1)
if (turn == 0): # turn left
# we turn the wheels right since
# we are backing up
RI.wheelsRight()
else:
# turn right
# we turn the wheels left since
# we are backing up
RI.wheelsLeft()
time.sleep(0.5)
RI.motorBackward(100,2.00)
time.sleep(0.5)
RI.wheelsMiddle()
```

Umieściliśmy robota w pokoju z meblami i złożonym zestawem ścian i puściliśmy go wolno. Oto wyniki z konsoli:

```
Simple Robot Brain
Continue straight =115.44cm
Continue straight =108.21cm
Continue straight =101.67cm
Continue straight = 95.67cm
```

Continue straight = 88.13cm

Continue straight = 79.85cm

Continue straight = 70.58cm

Continue straight = 63.89cm

Continue straight = 54.36cm

Continue straight = 44.65cm

Continue straight = 36.88cm

Continue straight = 28.32cm

Continue straight = 21.10cm

distance close enough so turn = 11.33cm

Turn Count = 1

Continue straight = 33.75cm

Continue straight = 25.12cm

distance close enough so turn = 18.20cm

Turn Count = 1

Continue straight = 40.51cm

Continue straight = 33.45cm

Continue straight = 24.73cm

distance close enough so turn = 14.83cm

Turn Count = 1

Continue straight = 35.72cm

Continue straight = 26.13cm

distance close enough so turn = 18.56c

Turn Count = 1

Continue straight = 43.63cm

Continue straight = 37.74cm

Continue straight = 27.33cm

Continue straight = 84.01cm

Możesz zobaczyć, jak robot zbliża się do ściany, a następnie obraca się kilka razy, aby znaleźć wyjście, a następnie kontynuuje jazdę. w filmie tutaj: https://youtu.be/U7_FJzRbsRw.

Lepsza architektura mózgu robota

Jeśli spojrzysz na oprogramowanie robotBrain.py z perspektywy architektury oprogramowania, jedna rzecz rzuca się w oczy. Główną częścią programu jest pojedyncza pętla while, która odpytuje czujnik (czujnik ultradźwiękowy), a następnie wykonuje jedną czynność naraz (ruchy, obroty itd.), a następnie ponownie odpytuje. Prowadzi to do nieco szarpanego zachowania robota (przesuń się trochę, wyczuj, przesuń się trochę, wyczuj itd.). Chociaż jest to najprostsza architektura, jakiej moglibyśmy użyć w naszym przykładzie, istnieją lepsze, choć bardziej skomplikowane, sposoby na zrobienie tego, które wykraczają poza zakres naszego dzisiejszego projektu. Te lepsze architektury są oparte na tak zwanych wątkach. Możesz pomyśleć o wątkach jako oddzielnych programach, które działają w tym samym czasie i komunikują się ze sobą za pomocą rzeczy zwanych semaforami i kolejkami danych. Semafory i kolejki danych to po prostu metody, za pomocą których wątek może bezpiecznie komunikować się z innymi wątkami. Ponieważ oba wątki działają w tym samym czasie, musisz uważać na to, jak rozmawiają i wymieniają informacje. Nie jest to skomplikowane, jeśli przestrzegasz zasad. Lepsza architektura dla naszego mózgu robota wyglądałaby tak:

Wątek silnika: ten wątek steruje silnikami. Sprawia, że działają one na polecenie i mogą zatrzymać silniki w dowolnym momencie.

Wątek czujnika: Ten wątek okresowo odczytuje czujnik ultradźwiękowy (i wszelkie inne czujniki, które możesz mieć), dzięki czemu zawsze masz dostęp do aktualnej odległości.

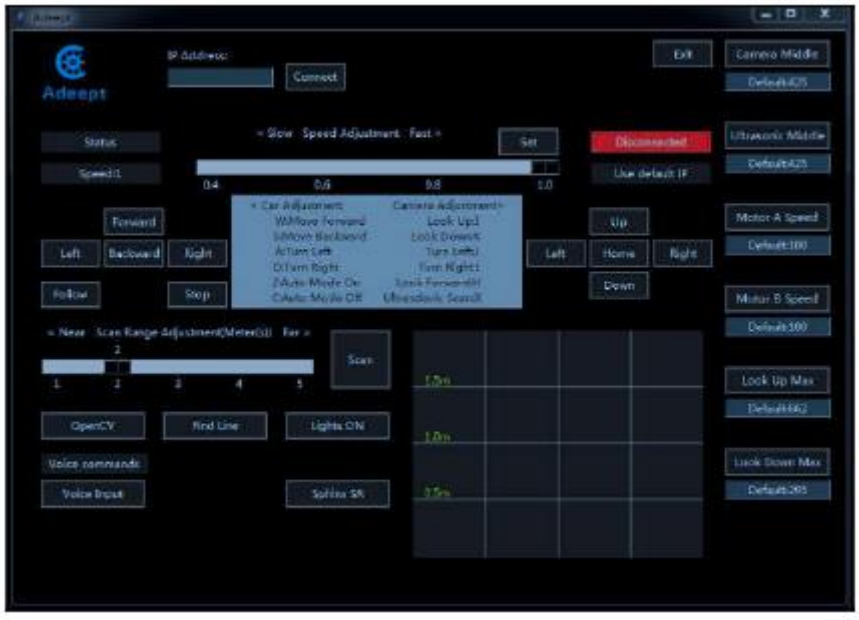
Wątek głowicy: Ten wątek steruje serwomechanizmami głowicy za pomocą poleceń z wątku poleceń.

Wątek polecenia: To jest mózg oprogramowania. Pobiera bieżące informacje z wątku czujnika i wysyła polecenia do silników i kieruje je do odpowiedniego wątku.

Taka architektura prowadzi do znacznie płynniejszej pracy robota. Możesz mieć uruchomione silniki podczas jednoczesnego pobierania wartości z czujników i wysyłania poleceń. Jest to architektura używana w oprogramowaniu Adeept server.py Plik dołączony do PiCar-B.

Przegląd dołączonego oprogramowania Adeept

Oprogramowanie Adeept dostarczane z robotem



to przede wszystkim model klient/serwer, w którym klient jest panelem kontrolnym na innym komputerze, a serwer działa na Raspberry Pi na PiCar-B. Panel kontrolny umożliwia zdalne sterowanie robotem i posiada wiele ciekawych funkcji, takich jak śledzenie obiektów za pomocą OpenCV oraz podobne do radaru możliwości mapowania ultradźwiękowego. Możesz także zobaczyć wideo pochodzące z robota i użyć go do ręcznej nawigacji.

Instalacja jest jednak dość skomplikowana, dlatego należy uważnie zapoznać się z instrukcjami. Oprogramowanie jest zdecydowanie przyjemne w użyciu, ale nie wymaga żadnego prawdziwego programowania. Całe oprogramowanie jest open source, więc możesz zajrzeć do środka, aby zobaczyć, jak sobie radzą. W szczególności sprawdź server.py w katalogu serwera i zobacz, w jaki sposób używają wątków, aby uzyskać płynny ruch robota.

Dokąd się stąd udać?

Masz teraz małego robota, który może wyświetlać dość złożone zachowanie w oparciu o wbudowany czujnik ultradźwiękowy. Możesz wiele dodać do tego robota, jeśli chodzi o dodawanie czujników do Raspberry Pi. (Co powiecie na pomiar odległości laserem? Czujniki zderzaka? Warunki oświetleniowe?) Wróć do Księgi 6 i połącz niektóre silniki i czujniki, których tam użyłeś, z tym robotem. Ponieważ wybraliśmy PiCar-B, możesz podłączyć Pi2Grover na górze sterownika silnika, aby móc korzystać ze wszystkich zgromadzonych urządzeń Grove. Niebo jest granicą!