

Practicumverslag Robotica

Gert Jan Verhoog

January 7, 2000

Abstract

Dit artikel beschrijft de hard- en software voor de Mindstorms robot die gemaakt is in het kader van het practicum behorend bij het vak Robotica, gegeven in het najaar van 1999 aan de faculteit Informatica, Universiteit Utrecht. Dit verslag behandelt het ontwerp van de robot en de problemen die zich voordeden bij het implementeren hiervan. De groep die dit practicum gemaakt heeft bestaat uit de volgende personen: Erwin Janssen, Patrick de Groot, Michiel Hermsen en Gert Jan Verhoog.

Contents

1	Beschrijving van de robot	2
1.1	Stevigheid van het ontwerp	2
1.2	Sensoren	3
1.3	Zes sensoren, drie sensor-inputs	3
2	Beschrijving van de software	4
3	De Utility componenten	5
3.1	statusregister	5
3.2	motorcontrol	5
4	De Behaviours	5
4.1	AvoidCollision	5
4.2	GripperToggle	5
4.3	RandomWalker	6
4.4	LightSeeker	6
4.5	SensorDecoder	6
4.5.1	Rechter Rotatie en linkerbumper	7
4.5.2	Linker rotatie en naar beneden gerichte lichtsensor	8
4.5.3	Rechter bumper en vooruit gerichte lichtsensor	9
4.5.4	Afstand en hoek	9

5	Problemen met deze aanpak	10
5.1	Onnauwkeurige sensoren met beperkte resolutie	11
5.2	Van elegante software naar quick & dirty hack	11
6	Een andere aanpak voor de software	11
7	Wat doet het wel, wat doet het niet	12
8	Conclusies	13
9	De samenwerking	14
9.1	Problemen met het practicum volgens Michiel	15
9.1.1	De Opdracht	15
9.1.2	De Samenwerking	15
9.1.3	De tijdsdruk	15
9.1.4	Het verschil met de collegies	16
9.1.5	Conclusie	16
9.2	Overzicht taakverdeling	16
9.2.1	Nawoord	17
9.2.2	Sources	17

1 Beschrijving van de robot

De lego-robot moet in staat zijn in een veld omringd door muren van bakstenen witgeverfde blikjes cola op te sporen, ze op te pakken en weg te brengen naar een door zwarte lijnen gemarkeerd "thuishonk". Dit legt een aantal ontwerp-eisen vast, waar ik hieronder op in zal gaan.

1.1 Stevigheid van het ontwerp

De robot moet stevig genoeg zijn om het gewicht van een (leeg) frisdrankblikje op te tillen met een grijpmechanisme. Ook moet het robotje bestand zijn tegen botsingen met obstakels en muren, en eventueel met andere robots. Dit is gerealiseerd door het robotje compact te bouwen, en de vitale onderdelen in een soort kooiconstructie vast te zetten. Dit bleek zeer goed te werken: De robot was compact en stevig, en je kon de robot aan (bijna) elk willekeurig onderdeel vastpakken en optillen zonder de robot te beschadigen. Ook tijdens de testrun presteerde het robotje voor wat stevigheid betreft uitstekend. Helaas was dit tijdens de eind-run van het practicum niet meer het geval: De motortjes kwamen soms los van hun ondergrond waardoor de aandrijving niet goed meer functioneerde. Ook daarna heeft het robotje bij mij thuis niet meer zo goed gepresteerd als eerder het geval was. Wat hier de oorzaak van is, heb ik niet kunnen achterhalen.

Het robotje heeft uiteindelijk twee grote wielen gekregen, elk apart aangedreven door een motor, en een glad legosteentje als glijdertje. De robot heeft een zeer stevige grijparm gekregen, die met één motor zowel de grijper kon openen en sluiten als de arm omhoog tillen. Door rubber bandjes aan de grijper te bevestigen, heeft de robot precies genoeg grip om het blikje op te tillen.

1.2 Sensoren

Een groot probleem bij alle robots is het gebrek aan sensorische input. Ook op robots als de Pioneer I en II in het robolab, waar — zeker in vergelijking met de sensoren van de legorobot — geavanceerde sensoren op zitten zoals sonar, een camera, laser-scanner en nauwkeurige rotatie-decoders blijkt dat je altijd meer sensoren zou willen gebruiken.

De robot moet blikjes kunnen herkennen, waarvoor een lichtsensor gebruikt wordt (een fotodiode met een rode led om objecten te beschijnen). Ook de zwarte markering op de vloer wordt herkend met een lichtsensor die naar beneden gericht is. Obstakels worden waargenomen met twee bumpers aan de voorkant van de robot, die een druksensor activeren op het moment dat de robot ergens tegen aan rijdt. De robot rijdt voornamelijk vooruit, daarom zitten de twee bumpers aan de voorkant. Op die manier kan de robot "voelen" aan welke kant een obstakel in de buurt is, links of rechts.

Om de robot de weg naar huis te laten vinden moet het op een of andere manier een idee hebben over de lokatie van zichzelf. Dit gebeurt door *dead-reckoning*: De informatie over de afstand en de hoek waarover de robot rijdt wordt gehaald uit rotatiesensoren die op de wiel-assen gemonteerd zijn. In de software worden die rotaties omgerekend naar afstand en hoek, zodat de robot zijn plaats kan bepalen.

1.3 Zes sensoren, drie sensor-inputs

Uit bovenstaande beschrijving blijkt dat er zes sensoren nodig zijn om de robot van voldoende informatie te voorzien voor het volbrengen van de opdracht. Het probleem is dat er op de lego-robot maar drie sensor-inputs beschikbaar zijn. Het is goed mogelijk om bijvoorbeeld een lichtsensor en een druksensor te combineren, maar het was een stuk moeilijker om een rotatiesensor te combineren met iets anders. Ook op het internet was er niemand die dit voor elkaar gekregen had. Toch bleek dit mogelijk te zijn door gebruik te maken van een complexe sensor-decoder, die op pagina 6 beschreven wordt. De volgende gegevens willen we uiteindelijk kunnen afleiden uit de sensor inputs:

1. **leftWheel** en **rightWheel**, tellertjes die het aantal rotaties van het linker- en rechterwiel aangeven;
2. **leftBumper** en **rightBumper**, booleans (statusbits) die aangeven of een druksensor is geactiveerd
3. **dLight** en **fLight**, de waarden van de lichtsensoren;
4. **heading**, de hoek ten opzichte van de beginhoek (de hoek die de robot had op het moment dat het programma wordt gestart wordt op 0 gezet).

2 Beschrijving van de software

Voor een kleine robot die gemaakt is om een betrekkelijk eenvoudige taak uit te voeren waar geen hogere cognitieve processen voor nodig zijn, is het handig om de software *behaviour based* te maken. Het gedrag van de robot wordt gedefinieerd in kleine, parallel draaiende *behaviours*, die allemaal een klein stukje van het totale gedrag uitvoeren. In veel behaviour-based robots heb je bijvoorbeeld een "avoid collision" behaviour of een "random-walk" behaviour.

De robot moet vaak dingen tegelijkertijd doen, en de behaviour-based manier van programmeren maakt dat op een elegante manier mogelijk. Een groot voordeel van deze manier van programmeren is dat de behaviours redelijk onafhankelijk van elkaar gemaakt kunnen worden, wat het programmeren makkelijker maakt: Als je een behaviour wilt testen, zet je gewoon andere behaviours uit.

De behaviours die in onze software gedefinieerd zijn zijn de volgende:

- **avoidcollision** grijpt in als de robot ergens tegenaan rijdt.
- **GripperToggle** opent en sluit de grijparm.
- **sensordecoder** decodeert de gecombineerde sensor-inputs en maakt er makkelijk interpreteerbare waarden van.
- **lightseeker** draait de robot naar de meest heldere lichtbron en schakelt op het juiste moment de GripperToggle-behaviour in.
- **randomwalker** rijdt willekeurig rond in het veld.

Deze behaviours zijn gemaakt in NQC's tasks. Tasks draaien in een eigen thread parallel met andere tasks. Suspend en Resume van tasks is geïmplementeerd door de statements `start task` en `stop task`. Andere onderdelen van de software zijn:

- **turndegrees** draait een bepaald aantal graden
- **drivedistance** rijdt een bepaalde afstand

Tenslotte is er nog een aantal utility functies gedefinieerd om het programmeren zelf wat eenvoudiger te maken.

- **statusregister** houdt in 2 integers maximaal 32 boolean waarden bij.
- **motorcontrol** bestaat uit een aantal macros die de motoren aan en uit zetten, en tegelijk *state* bewaren over de toestand van de motoren.

In de hier volgende hoofdstukken zal ik uitgebreider stilstaan bij de gebruikte software componenten.

3 De Utility componenten

3.1 statusregister

Statusregister bewaart in twee integers maximaal 32 boolean waarden. Hierdoor wordt het aantal variabelen dat gedeclareerd moet worden drastisch verkleind. Iets dat onder NQC heel hard nodig is omdat het maar 32 variabelen toestaat.

Met `#define`'s zijn een aantal commando's gedefinieerd waarmee individuele bits of bitpatronen te zetten of te *clearen* zijn. De commando's zijn `isSet(v)`, `setBit(v)` en `clearBit(v)`. Voor de bits die kunnen worden gemanipuleerd zijn constanten gedefinieerd, zoals *leftMotorOn*, *gripperIsMoving*, etcetera.

3.2 motorcontrol

Motorcontrol bestaat uit een aantal commando's die de motoren vooruit en achteruit kunnen laten draaien of stil kunnen zetten. Deze commando's roepen de NQC-statements `OnFwd`, `OnRev`, `Off` en dergelijke aan, maar ze zetten ook een aantal statusbits zodat de rest van de software weet of een motor vooruit of achteruit rijdt, of dat de motor stil staat. Ook wordt de *laatst gebruikte draairichting* bewaard. Dit is nodig omdat wanneer de motor gestopt wordt (door het gebruik van `Off` of `Float`), hij niet meteen stilstaat maar nog een stukje verder draait. Voor de sensordecoder is het echter belangrijk om te weten welke kant een motortje op draait, dus moet deze state-informatie bewaard worden.

4 De Behaviours

4.1 AvoidCollision

AvoidCollision kijkt of er één of twee druksensoren van de bumpers zijn geactiveerd. Als dit het geval is, wordt de statusbit **avoidCollisionActive** gezet. Andere tasks kunnen daar dan (eventueel) rekening mee houden. De robot stopt de motoren en rijdt een stukje achteruit. Als de linkerbumper actief is, draait de robot vervolgens met een willekeurige hoek naar rechts; als de rechterbumper actief is naar links, en als ze allebei actief zijn (de robot is dan recht op een obstakel ingereden) draait de robot een willekeurige hoek in een willekeurige richting. De statusbit **lastTurnWasRight** wordt gezet als de robot naar rechts is gedraaid.

4.2 GripperToggle

De GripperToggle task wordt normaal gesproken alleen aangeroepen door het commando `doGripperToggle`, die de gripper opent of sluit, afhankelijk van de vorige status van de gripper. De GripperToggle task wordt gestart, en na een constante tijd (de tijd die het kost de gripper te openen of te sluiten) wordt de task gestopt. Terwijl de grijparm in

beweging is, staat de statusbit **gripperIsMoving** op 1, zodat andere tasks weten dat de gripper bezig is.

4.3 RandomWalker

De Randomwalker task doet niets anders dan een willekeurige afstand vooruit rijden, daarna over een willekeurige hoek draaien en vervolgens weer een willekeurige afstand rijden. In eerste instantie hadden we een andere manier bedacht om het veld te doorkruisen op zoek naar blikjes. Dat was een behaviour die zigzaggend heen en weer reed, waarbij de hoek die een bepaalde kant op gedraaid werd elke keer afhankelijk was van de lichtintensiteit van de vooruitkijkende lichtsensor. Op die manier zou de amplitude van de slingerbeweging kleiner worden en zou het gedrag convergeren naar het rechtdoor rijden, waarbij op geringe afstand van de robot recht vooruit een blikje zou staan. De praktijk wees echter uit dat de sensoren niet nauwkeurig genoeg waren voor dit gedrag.

4.4 LightSeeker

De LightSeeker kijkt naar de intensiteit van het opgevangen licht. Als de intensiteit minder wordt, dan wordt de bit **lightSeekerActive** gezet, en draait de robot een stukje terug. Om te weten in welke richting de robot terug moet draaien, wordt de status van **lastTurnWasRight** gelezen.

Als de lichtintensiteit gelijk is aan of hoger is dan de **canInSight** variabele, besluit LightSeeker om het blikje te pakken. De waarde van **canInSight** wordt aan het begin van een sessie gezet door het blikje vlak voor de robot te houden en vervolgens de linkerbumper te activeren. Op dat moment wordt de gemeten lichtintensiteit bewaard in **canInSight**. Bij het bepalen of er in een bepaalde richting meer licht te zien is werd eerder gebruik gemaakt van het verschil tussen de huidige lichtwaarde en het gemiddelde van de daarvoor gemeten waarden (het gemiddelde omgevingslicht), maar dit bleek niet goed te werken.

4.5 SensorDecoder

Op onze robot zijn de sensoren op de volgende manier aangesloten:

1. Rechter rotatie & linkerbumper
2. Linker rotatie & naar beneden gerichte lichtsensor
3. Linker bumper & vooruit gerichte lichtsensor

Voor deze configuratie is gekozen omdat de beide lichtsensoren onafhankelijk uitgelezen moesten worden, in verband met de oneffen gekleurde vloer, zodat er dus in ieder geval één lichtsensor met een rotatiesensor gecombineerd moest worden. Omdat door de ingewikkelde codering van de input signalen waarschijnlijk wat nauwkeurigheid verloren zou gaan is gekozen om de naar beneden gerichte lichtsensor met

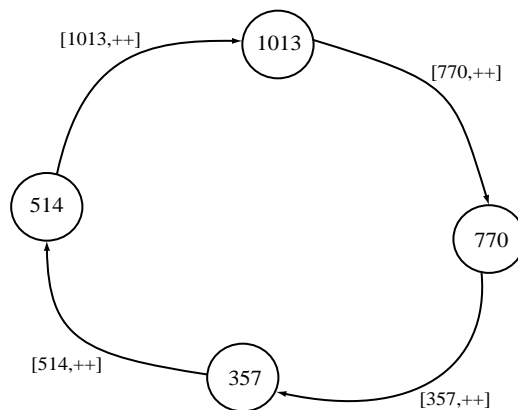


Figure 1: Een deel van een statemachine om rotatiesignalen te decoderen. De waarden van de states zijn de waarden van de rotatiesignalen. De '++' betekent: Tel 1 op bij de huidige waarde van de rotatie.

een rotatiesensor te combineren, en de vooruit gerichte sensor met een bumper. Dan blijven er nog een rotatiesensor en een bumper over.

Op het WWW was geen informatie te vinden over de combinatie van deze sensoren, met name de rotatiesensor in combinatie met iets anders werd nergens beschreven, of er werd gezegd dat het niet mogelijk was. Ik zal eerst de soorten signalen beschrijven die de verschillende sensoren leveren, daarna zal ik uitleggen hoe wij alle sensordata kunnen verwerken.

4.5.1 Rechter Rotatie en linkerbumper

Deze sensoren, die allebei op input 1 zijn aangesloten, worden vertaald in een waarde voor een wieltellertje, **rightWheel**, en een statusbit, **leftBumper**.

De rotatiesensor, die werkt met een optische decoder zoals de wieltes in een muis, geeft vier verschillende waarden terug: {1013, 770, 357, 514}. Wanneer je eerst de waarde 1013 terug krijgt en daarna 770 weet je bijvoorbeeld dat het wiel vooruit is gedraaid. Als je in plaats van 770 514 terugkrijgt weet je dat het wiel achteruit is gedraaid. Op die manier is een finite state machine te bedenken die de rotatiesignalen omzet naar een tellertje dat opgehoogd of verlaagd wordt. (zie **figuur 1**).

Een probleem dat we tegen kwamen is dat niet elke "stap" van de rotatiesensor even lang is: Met onze wielomtrek wordt met elke stap 15mm afgelegd, behalve bij de stap met de laagste waarde, 357: dan is de afgelegde afstand ineens 23mm, bijna anderhalf keer zoveel! Aangezien NQC als enige datatype integers heeft, hebben we besloten het wieltellertje overal met 2 op te hogen, en bij de grote stap met 3.

Als de bumper is geactiveerd, geeft de sensor input een waarde kleiner dan 100 terug.

Een groot probleem dat we tegenkwamen bij het decoderen van

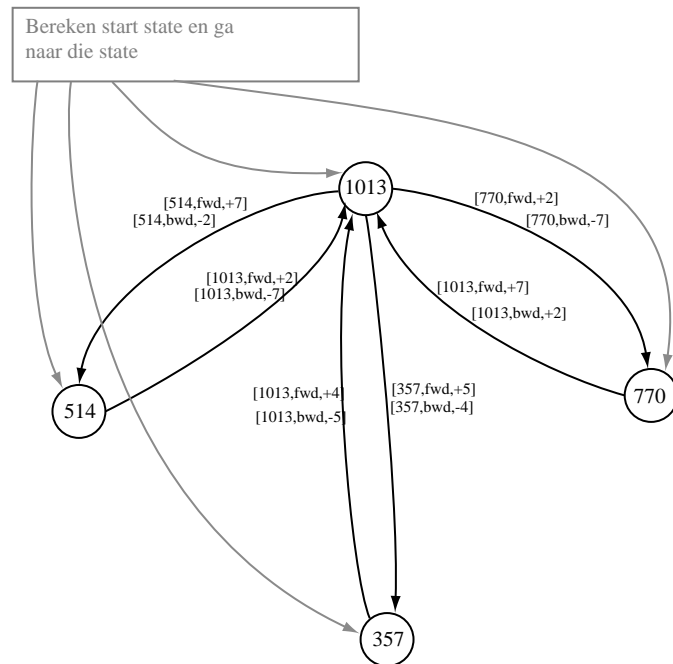


Figure 2: Een deel van een ingewikkelder state machine voor de rotaties. In deze tekening is alleen de transitie van en naar state 1013 compleet getekend, voor de overige state-transities werkt het op dezelfde manier. NB: In dit figuur is nog geen rekening gehouden met de bumper.

de sensorwaarden is de traagheid van NQC: Soms draait een wiel *meer dan één stap* voordat de rotatie-decoder aan een nieuwe iteratie begint. Bij de eerste versie van de decoder raakte de state machine in de war. De volledige state machine is daarom behoorlijk ingewikkeld, zoals te zien is in **figuur 2**, omdat er niet alleen pijlen lopen van de huidige state naar de vorige en volgende states, maar naar alle andere states.

Eerst moet naar de rotatiewaarde worden gekeken, zodat bepaald kan worden in welke state de decoder moet starten. Als de bumper wordt geactiveerd, moet — na deactivering van de bumper — de state weer berekend worden: Het wiel kan nog wat gedraaid zijn op het moment dat de bumper is geactiveerd, maar dat kan de decoder niet zien!

4.5.2 Linker rotatie en naar beneden gerichte lichtsens

De decodering van input twee naar een waarde voor **rightWheel** en **dLight** (Downward Looking Light) gaat ongeveer op dezelfde manier

stand	rotatie	rotatie+licht
1	1013	680 – 1024
2	770	430 – 570
3	357	280 – 350
4	514	300 – 350

Figure 3: De vier mogelijke waarden die de rotatiesensor door kan geven

als met de andere rotatiesensor die hierboven is beschreven, alleen is er nog een extra moeilijkheid: De lichtsensoren geeft een waarde af tussen de 480 en 800. Wanneer de licht- en rotatiesensoren gecombineerd worden, blijken voor elke stand van de rotatiesensor de waarden binnen een bepaalde range te liggen, zoals te zien is in **tabel 3**.

Uit **tabel 3** blijkt ook dat de laatste twee rotatiestanden, 357 en 514, overlappende ranges opleveren. De laatste twee states van de decoder worden dus bij elkaar gevoegd, omdat er toch geen onderscheid te maken is. De waarde voor het licht wordt dan ($value - rangeStart$). Dit blijkt in de praktijk redelijk goed te werken.

4.5.3 Rechter bumper en vooruit gerichte lichtsensoren

De vertaling van input 3 naar de statusbit **rightBumper** en de variabele **flLight** (Forward Looking Light) is, in vergelijking met de vorige twee inputs, triviaal: Als de bumper geactiveerd is, is de waarde kleiner of gelijk aan 100. Anders is de waarde precies de lichtintensiteit van de sensor.

4.5.4 Afstand en hoek

De afstand is het verschil tussen een begin- en eindpositie:

$$d = (lWheelBegin + rWheelBegin)/2 - (lWheelEnd + rWheelEnd)/2.$$

Voor de hoek is wat meer rekenwerk nodig. Als de wielen met ongelijke snelheid draaien, beschrijven beide wielen een cirkel met verschillende diameter, maar met hetzelfde middelpunt. (Zie **figuur 4**). Merk op dat de afstand tussen beide wielen 10.5cm bedraagt.

Afstand d_2 is te schrijven als $\alpha(R_1 + R_2)$, afstand d_1 als αR_1 . Dan geldt $d_2 = \alpha(10.5 \frac{d_1}{\alpha}) = \alpha 10.5 + d_1$. Uiteindelijk is af te leiden dat geldt $\alpha = \frac{d_2 - d_1}{10.5}$, gemeten in radialen.

Op deze manier kan de heading dus bijgehouden worden, en op die manier kan de functie **turnDegrees** zien hoe ver gedraaid moet worden.

Het grote voordeel van deze manier van afstand en hoek bepalen is dat het rijden en draaien niet op tijd hoeft te gebeuren: Normaal gesproken zeg je "draai gedurende 1 seconde rond", maar dat heeft als nadeel dat je nooit precies weet hoever je draait: Dat is afhankelijk van de ondergrond, of de robot een blikje vast heeft (extra gewicht) en andere factoren. De functie om over een bepaalde afstand te draaien

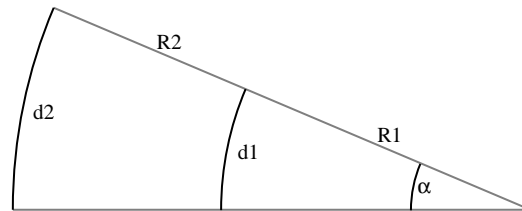


Figure 4: De wielen van de robot beschrijven bij een bocht beide een cirkelboog. Wiel 1 (het rechterwiel) beweegt over een cirkelboog met lengte $d1$ en straal $R1$, wiel 2 over een lengte $d2$ met straal $R1+R2$

werkte beter: Als de robot tegengehouden wordt tijdens het draaien, ging hij na loslaten van de robot weer gewoon verder met draaien.

De heading wordt ook gebruikt in de functie `driveDistance`. Als de heading tussen begin en eind van de rit verandert, betekent dat dat de robot niet in een rechte lijn rijdt. `driveDistance` zet dan één van de wielen op *float* met `TurnFloat(...)`, totdat de heading weer gelijk is aan de beginwaarde.

Uiteindelijk hebben we dit niet kunnen gebruiken in de uiteindelijke software. Eén van de problemen waar we mee te maken hadden is de traagheid van NQC: De sensordecoder liet, zelfs in z'n meest ingewikkelde en complete vorm, wel eens een steekje vallen. `TurnDegrees` werkte het best bij hoeken van 360 graden. Bij een test waarbij de robot voortdurend 360 graden om z'n as draaide, bleek dat hij meestal wel *ongeveer* een hele cirkel draaide, maar soms maar een kwart cirkel. Dit probleem hebben we er niet uit kunnen krijgen. `DriveDistance` had dezelfde soort problemen.

Deze problemen worden veroorzaakt door een enorm probleem in NQC: Een gebrek aan floating- of fixed point operaties. Hierdoor moesten bij sommige berekeningen getallen vermenigvuldigd worden om overal integers uit te krijgen, maar de afrondingsfout was daardoor erg groot.

5 Problemen met deze aanpak

In theorie hadden we elegante software geschreven, met verschillende behaviours die mooi in elkaar grepen. Deze behaviours maakten gebruik van de berekende hoek en afstand, zodat het systeem op een mooie manier kon navigeren en bewegen. Voor deze aanpak is gekozen omdat we ervaring wilden krijgen in het op deze manier programmeren van mobiele robots, één van de doelstellingen achter het practicum.

Helaas zijn we door een aantal problemen genoodzaakt geweest onze aanpak te veranderen.

5.1 Onnauwkeurige sensoren met beperkte resolutie

De sensoren waren niet bepaald nauwkeurig. In **tabel 3** beschreef ik de waarden die een rotatiesensor kon hebben, maar in de praktijk wijken deze waarden soms wel met 5 af van de nominale waarde. Ook lichtsensoren geven een "onrustig" beeld: Ook al staat de lichtsensor stil en gericht op een object dat ook stilstaat, in een statische omgeving, dan nog fluctueert de waarde van het licht tussen elke meting met een grootte van tussen de 1 en de 4.

De resolutie van de rotatiesensoren was zeer laag: Eén state-transitie van de rotatiesensor komt overeen met minstens 1.5cm afstand. In een eerdere versie van de robot zat er een versnelling tussen de aandrijfassen en de rotatiesensor, maar daar was NQC veel te langzaam voor, dus we hadden geen alternatief.

Het bereik van de lichtsensor was zeer beperkt. Op een afstand van meer dan 5cm was het verschil in lichtintensiteit tussen wel een blikje en geen blikje al minder dan de natuurlijke variatie in waarden die de lichtsensor gaf. Dit maakt het robotje praktisch blind. Onze eerdere strategie, om zigzaggend met een afnemende uitwijking op het blikje af te rijden, kon daarom niet werken.

Door deze onnauwkeurigheden en door het gebrek aan floating- of fixed-point math in NQC was het daardoor erg moeilijk om nauwkeurig de heading en de afstand te berekenen. Uiteindelijk moesten we constateren dat we deze variabelen een minder belangrijke rol moesten geven in onze software. Om die reden moesten we afstappen van het idee dat we de bewegingen van de robot *niet* met tijd controleerden maar met echte hoeken en afstanden.

5.2 Van elegante software naar quick & dirty hack

Na de testrun bleek dat we met onze aanpak niet ver zouden komen. Het was tijd voor een andere aanpak: De *quick and dirty hack*. Hoewel ik de educatieve waarde van zo'n hack niet zo groot vind als de elegantere oplossingen die we eerder bedacht hadden, leek dit de enige oplossing om het practicum nog een kans van slagen te geven.

6 Een andere aanpak voor de software

Inmiddels had ons project aardig wat vertraging opgelopen omdat we te laat in zagen dat de sensoren en de taal NQC zo beperkt waren dat ons idee niet gerealiseerd zou kunnen worden. Daarom hebben we geprobeerd zo snel mogelijk nog werkende software in elkaar te zetten.

Het belangrijkste verschil met de oude software is dat we de bewegingen van de robot — rijden over een bepaalde afstand en draaien — nu *wel* met tijd doen. Hierdoor hebben we de manier waarop de

robot door het veld rijdt moeten versimpelen: De robot rijdt nu rond met de task **RandomWalker**. De robot rijdt voor een willekeurig aantal milliseconden, draait dan willekeurig door één van de motoren een willekeurig aantal milliseconden stop te zetten en rijdt vervolgens weer voor een willekeurig aantal milliseconden. De task **AvoidCollision** zorgt dat de robot niet vastloopt als hij tegen een obstakel rijdt.

De task **lightSeeker** probeert op een slimme manier bij te draaien in de richting van het helderste licht, en grijpt daarom in op **RandomWalker** zodra het licht zwakker wordt (de robot draait dan weg van het licht en **lightSeeker** draait weer terug). Op het moment dat de lichtsensoren een aan het begin vastgestelde waarde, **canInSight** heeft, gaat hij er van uit dat er een blikje tussen de grippers staat en roept **DoGripperToggle** aan. Door de onnauwkeurigheid van de lichtsensoren werkt het bijdraaien op grotere afstanden van het blikje niet, zodat meestal gewoon random rond gereden wordt, en de **canInSight** waarde wordt vaak waargenomen, de robot ziet dus op de vreemdste momenten blikjes.

7 Wat doet het wel, wat doet het niet

De software zoals die nu werkt, staat hierboven beschreven. Eén routine schittert daarbij door afwezigheid: Hoe rijdt de robot terug naar het thuishonk om het blikje weg te zetten?

Gebruik makend van de heading en afstand hebben we nagedacht over een manier om het veld in coördinaten te verdelen. In de opdracht stond dat we bij de tweede run gebruik moesten maken van informatie uit de eerste run. Het ligt voor de hand om dan de lokatie van de blikjes op te slaan.

Omdat het veld in eerste instantie 2.5m X 2.5m was, konden we de resolutie op 1cm zetten, en zo twee coördinaten in de range 0..250 in één 16 bits integer zetten. Op die manier konden we ook de coördinaten van de gevonden blikjes opslaan voor een volgende run. Op die manier zou de robot mooi naar het thuishonk kunnen terug navigeren. Helaas was daar een goede hoek- en afstandsmeting voor nodig, en bovendien zou er dan met sinussen en cosinussen gerekend moeten worden. Dit is echter onmogelijk in NQC. We hebben dit dus niet verder uit kunnen werken. Een makkelijkere oplossing waar we over gedacht hebben, was om de muren te volgen en zo bij het thuishonk aan te komen, maar dit zou niet erg mooi werken, omdat er in eerste instantie *twee* thuishonken zouden zijn. Uiteindelijk is het niet meer gelukt een terugrijd-algoritme te implementeren.

De overige software levert ongeveer het volgende gedrag op: De robot rijdt willekeurig rond, botst af en toe ergens tegenaan en draait dan om, en opent/sluit af en toe z'n gripper. Het is wel zo dat een blikje dat pal voor de neus van de robot staat altijd leidt tot het oppakken er van.

8 Conclusies

NQC is een minder geschikte taal om serieuze of complexe robot-control software in te schrijven. Het gebrek aan verschillende datatypen, de beperking tot 32 variabelen en de traagheid doordat het compileert naar byte-code die door de firmware geïnterpreteerd wordt maakt het erg moeilijk complexe software te schrijven. Ook het feit dat de RCX maar drie inputs heeft maakt het lastig grotere robots te realiseren.

Een alternatief waar ik naar gekeken heb is legOS. Daarmee kun je in C of C++ schrijven en compileren naar Hitachi-8300 instructies die door de RCX direct uitgevoerd kunnen worden. Delen van de software die ik voor het Robocup project geschreven heb en nog aan het schrijven ben, zoals een eenvoudig te gebruiken en kleine op de POSIX-thread library pthread gebaseerde thread class, een eenvoudig framework voor het schrijven van behaviours en een representatiemodel om informatie over de buitenwereld in op te slaan, zouden — zonder wellicht een aantal voor het legopracticum overbodige componenten — gebruikt kunnen worden als basis voor de lego-software.

Helaas heeft de computer waar ik de H8300 gcc compiler op gebouwd heb en de legOS software op staat geen seriële poort en bestaan er geen legOS-download tools voor het OS (Mac OS) waar de IR-Tower *wel* op aangesloten is, dus heb ik dat niet uit kunnen proberen. De toolkit die we in het robolab ontwikkelen is gemaakt met het idee van platform- en applicatie-onafhankelijkheid in gedachten, en het porten van delen van de robocup software (de toolkit dan zonder het voetbalgedrag) naar legOS en de Mindstorms robot is iets dat ik nog wel eens zou willen proberen.

Door de beperkte resources en gebrekkige sensoren is één van de doelstellingen van het practicum heel goed over gekomen: Het programmeren van robots is *behoorlijk moeilijk*.

Wij denken dat deze opdracht eigenlijk te moeilijk was voor de beschikbare tijd. Het feit dat we van te voren niet hadden kunnen weten dat de sensoren zo gebrekkig waren heeft er voor gezorgd dat we veel dingen op een later tijdstip hebben moeten veranderen. Ik hoop dat ik in dit artikel heb laten zien dat dit niet kwam omdat we onze tijd verkeerd gepland hebben, maar omdat we de software in eerste instantie niet als quick & dirty hack wilden schrijven.

We vonden het jammer dat we bij het college behandelde theorieën over het kinematics probleem en motion planning niet in de praktijk konden brengen met deze opdracht. Hierdoor had het college niet echt iets met het practicum te maken. Wij vonden het practicum allemaal wel heel erg leuk. Hoewel we uiteindelijk de software als hack in elkaar hebben moeten zetten, waar je niet echt veel van leert, is het practicum toch leerzaam geweest: Door de beperkte middelen word je gedwongen oplossingen te bedenken voor problemen die in eerste instantie heel eenvoudig lijken, of waar je normaal niet bij stilstaat, zoals het bepalen van de heading uit rotatiestappen van de wielen.

9 De samenwerking

De indruk zou kunnen ontstaan dat onze practicum opdracht niet helemaal geslaagd is door verkeerde planning en gebrek aan inzet. Dit is wat mij betreft beslist niet waar, maar door een aantal oorzaken, waaronder het feit dat een te groot deel van het practicum voor rekening van één persoon moest komen is de tijdsdruk te groot geweest. Ik heb in de dagen na de testrun besloten me vooral op het leren van het tentamen te richten, omdat ik het gevoel had dat ik al voldoende tijd in het practicum had gestoken, en ik wilde absoluut het tentamen halen.

Met deze beschrijving van de taakverdeling en problemen met de samenwerking hoop ik één van de oorzaken van het niet goed slagen van onze opdracht uit te kunnen leggen.

De samenwerking binnen de groep verliep niet helemaal zoals het zou moeten, waardoor er niet op de meest efficiënte manier gewerkt kon worden. Met name Gert Jan heeft het idee dat hij naar verhouding veel meer gedaan heeft dan de andere teamleden. Omdat ik niet zonder meer mijn klachten over andere personen in dit verslag wil schrijven heb ik de andere teamleden gevraagd naar hun mening hierover.

Door drukke en incompatible agenda's van de teamleden konden we maar op één middag bij elkaar komen, donderdagmiddag. Op zich is dit geen probleem: Met het robocup project komen we ook maar één keer per week bij elkaar: om te overleggen, te testen en beslissingen te nemen, en het ontwerpen en coderen doen we voor het grootste gedeelte thuis of op andere dagen in het robolab. Informatie wordt gedurende de hele week uitgewisseld via e-mail.

Helaas bleek deze werkwijze voor ons practicumteam niet voor iedereen vanzelfsprekend. Op de donderdagmiddagen hebben we meerdere malen afgesproken waar iedereen zich de komende week mee bezig zou houden, en we zouden dan via e-mail contact houden en code uitwisselen (die ik dan kon testen omdat ik het robotje thuis had). Dit is niet gebeurd, waardoor te veel op de donderdagmiddag moest gebeuren. Niet alleen is 1 dag per week onvoldoende tijd om aan het practicum te besteden, met z'n vieren achter een computer zitten om te coden is erg inefficiënt. Bovendien bleek dat niet iedereen genoeg programmeerervaring had. Daardoor kwam het coderen, maar ook veel van het ontwerpen voor rekening van Gert Jan.

Van Erwin heb ik niets meer gehoord na de eind-run op de dag van het tentamen, maar met Patrick en Michiel heb ik contact gehad over de samenwerking in de groep. Zij bevestigen beiden dat ze in verhouding te weinig aan het practicum gedaan hebben en dat het meeste werk door mij is gedaan. Ze erkennen ook dat er meer thuis had moeten gebeuren, maar door een aantal oorzaken, zoals problemen met e-mail en computers, en een gebrek aan programmeer-ervaring is dit niet genoeg gebeurd.

Michiel heeft beschreven waar volgens hem de problemen met het practicum liggen in het volgende stuk:

9.1 Problemen met het practicum volgens Michiel

Er zijn verschillende oorzaken aan te wijzen van ons falen.

9.1.1 De Opdracht

Onze robot moest de omgeving af scannen naar blikjes en dan met steeds kleiner wordende zwenkbewegingen naar het blikje toe te bewegen. Het bleek al snel dat de lichtsensor de blikjes niet van veraf kon herkennen. Onze uiteindelijke oplossing hiervoor was het random bewegen en maar hopen dat je een blikje tegenkomt. Als dit gebeurt is moet er terug worden gereden naar het thuisvak. Hiervoor was het de bedoeling door middel van een (ruwe) plaatsbepaling richting het thuisvak te rijden. Hierbij hebben we niet gedacht aan simpele oplossingen, gedeeltelijk omdat de opdracht vroeg om een oplossing waarbij positiebepaling noodzakelijk leek.

In de opdracht was het de bedoeling dat bij een tweede run, de blikjes sneller gevonden zouden worden. De enige manier om dat te doen is, door te 'onthouden' waar de blikjes waren. Een vorm van plaatsbepaling is hiervoor noodzakelijk.

De opdracht werd aangepast zodat het technisch mogelijk zou zijn om de robot de opdracht te laten uitvoeren. Maar helaas waren wij al door dit voorval op het verkeerde been gezet, want wij zaten te denken over het bepalen van de coördinaten en richtingshoek van de robot, die wij zouden kunnen gebruiken om zo de posities van de blikjes (die we vonden) te onthouden en om onze weg terug naar de thuisbasis te vinden. Helaas is het bepalen van coördinaten niet mogelijk door de simpliciteit NQC; er was geen mogelijkheid tot het gebruiken van sinus of cosinus. Het bepalen van de richtingshoek van de robot zou wel kunnen werken, maar doordat de draaisensoren niet al te precies zijn, bleek dit ook onhaalbaar. We hebben veel tijd verspeeld aan deze problematiek, en hebben niet genoeg nagedacht over de simpele oplossingen. Uiteindelijk bleek tijdens de demonstratie dat het rijden langs de zijanten de enige (werkende) oplossing voor dit probleem te zijn. Over deze optie hadden we alleen op het laatst een korte discussie.

9.1.2 De Samenwerking

Doordat er enkele personen niet veel programmeerervaring hadden, kon er thuis niet veel geprogrammeerd worden. Ook door de verschillende roosters en werkschema's lagen afspraken in de weg.

9.1.3 De tijdsdruk

In week 5 (van de 7 in het blok) wisten we hoe de opdracht en de arena er uit zouden zien. Toen bleek pas echt dat het niet mogelijk was om met de robot te doen wat we van te voren bedacht hadden. Daarna hebben we in de week erop nog overleg gehad over het 'terugrijd-probleem', maar hier geen adequate oplossing voor gevonden. In de laatste week besloten we om ons op het tentamen te richten.

9.1.4 Het verschil met de colleges

De stof die behandeld werd bij de colleges bleek niet of nauwelijks toepasbaar in het praktikum, ook hierdoor was het voor de onervaren programmeurs soms moeilijk om het praktikum te maken. Er was geen houvast aan de theoretische stof.

9.1.5 Conclusie

Vooraf door het te hoge eisen stellen aan de robot hebben we de simpele oplossingen over het hoofd gezien. Onze eisen leken gelet op de opdracht zeer redelijk, maar de opdracht werd dan ook bijgesteld. We hadden aan het eind wel iets in elkaar kunnen flansen, maar doordat het tentamen ernstig naderde hebben we dit verzuimd. Al met al hebben we toch een redelijk idee kunnen vormen van hoe een robot een taak zoals de opdracht kan doen en wat daar voor nodig is.

9.2 Overzicht taakverdeling

Tot slot nog een overzicht van de dingen die iedereen gedaan heeft. Ik heb de andere teamleden gevraagd te zeggen wat ze van dit lijstje vonden en of dit een eerlijke weergave van de werkelijkheid is. Dit is door Michiel en Patrick bevestigend beantwoord, van Erwin heb ik niets gehoord.

Patrick: Heeft meegewerkt aan het ontwerpen van het rotatie-naar-hoek algoritme en aan de algemene structuur van het programma en beslissingen over het laten vallen van de `turnDegrees/turnDistance` dingen. Meegewerkt aan verschillende behaviours, zoals het zigzaggen. Heeft ook grammaticale wijzigingen in het practicumverslag gegeven.

Michiel: Heeft meegewerkt aan het algemene ontwerp en de structuur van het programma en beslissingen over het laten vallen van de `turnDegrees/turnDistance` dingen. Heeft een algoritme in NQC bedacht voor het omrekenen van rotaties naar de hoek. Meegewerkt aan verschillende behaviours, zoals het zigzaggen. Heeft een gedeelte voor het practicumverslag geschreven over de reden dat het practicum niet met groot succes is afgerond.

Erwin: Heeft het oorspronkelijke, zigzag-behaviour gemaakt, mee gedacht over de structuur van de software en heeft het wetenschappelijke artikel gemaakt.

Gert Jan: Heeft de robot ontworpen en gebouwd, heeft een eerste ontwerp voor de structuur van de software bedacht en verschillende onderdelen daarvan geïmplementeerd, heeft de utility-functies zoals statusregister en motorcontrol macros bedacht en geïmplementeerd, heeft de sensor-decoder voor het verwerken van 6 sensoren ontworpen en geïmplementeerd, heeft meegewerkt aan het bedenken van een manier om rotaties naar een draaihoek te converteren en het in een testversie geïmplementeerd (`turnDegrees`), heeft het practicumverslag gemaakt.

Iedereen heeft op de donderdag-bijeenkomsten meegewerkt en overlegd over de te volgen strategie.

9.2.1 Nawoord

Met dit verslag is duidelijk geworden wat de ontwerpbeslissingen zijn geweest van ons lego-robot project, wat de technische problemen waren en wat de oorzaken zijn van het niet met succes realiseren van een blikjes-ophalend lego robotje. Ook is duidelijk geworden dat de teamleden serieus gewerkt hebben aan het practicum, en dat sommige teamleden zeer veel tijd en moeite in de opdracht geïnvesteerd hebben.

Ondanks het feit dat ons team er niet in geslaagd is op tijd een goed werkende robot te demonstreren hoop ik dat uit dit verslag duidelijk wordt dat wij aan de eisen van het practicum, het leren omgaan met de specifieke moeilijkheden van het programmeren van autonome mobiele robotjes door het ontwerpen en implementeren van software en hardware voor het opsporen en verzamelen van frisdrankblikjes, toch in voldoende mate voldaan hebben.

9.2.2 Sources

De laatste sources van het legorobotje, alsmede de PDF files van dit verslag en het artikel van Erwin zijn op het World Wide Web te vinden in de volgende directory: <http://www.xs4all.nl/gerti/canbot/>