Sphero Corridor Mapping

Mapping op basis van collision detectie met een robotische bal

Tom Koenen & Lotte Weerts

28 juni 2013

Inleiding

In dit onderzoek wordt getracht om van een onbekende ruimte een plattegrond te maken met behulp van een robot. Het beteft een robotische bal, de Sphero van fabrikant Orbotix (zie Appendix A), die op basis van het detecteren van *collisions* (botsingen met objecten of muren) een interne representatie van een ruimte moet gaan genereren. Op basis van deze representatie zou de robot vervolgens in staat moeten zijn om zonder botsingen door de ruimte te bewegen.

Het autonoom verkennen van een ruimte is toe te passen in verschillende situaties, bijvoorbeeld onbemande robots in rampsituaties. Deze robots moeten in staat zijn een ruimte te verkennen en een plattegrond te maken van de ruimte. Wanneer deze robots ook als taak hebben slachtoffers mee te nemen uit het gevaarlijke gebied, is het veilig terugkeren naar het startpunt van levensbelang. Hieruit blijkt de maatschappelijke relevantie van dit onderzoek.

Het maken van een plattegrond van een onbekende ruimte wordt in de literatuur gedefinieerd als SLAM: Simultaneous Locating and Mapping (Thrun, 2005). Het betreft hier een kip- en ei probleem: om een plattegrond te maken moet de robot weten waar hij is, en om te weten waar hij is heeft hij een plattegrond nodig. Een veelgebruikt algoritme in SLAM is het *Kalman filter* algoritme (Thrun, 2005). Dit algoritme maakt onder andere gebruik van *landmarks*, herkenbare objecten in een ruimte. Echter, collisions zijn ongeschikt als landmarks. Dit komt doordat het bijna onmogelijk is om twee collisions als zijnde hetzelfde object aan elkaar te koppelen. Het is namelijk niet eenduidig te zeggen of twee verschillende datapunten feitelijk hetzelfde object zijn maar een beetje afwijken door inaccurate localisatie, of dat ze twee losse of misschien zelfs een groot object representeren. De Kalman filter lijkt dus niet geschikt voor dit onderzoek.

Omdat de Sphero al beschikt over een interne localisatiemethode ligt het probleem bij dit onderzoek met name in de *mapping*, het verwerken van de datapunten tot een plattegrond. Een methode die hiervoor geschikt lijkt is *sum of least squares data fitting* (Bretscher, 2009). Hiermee kunnen namelijk rechte lijnen door data getroken worden, wat gebruikt zou kunnen worden voor het detecteren van bijvoorbeeld muren.

Eerder werd al gesteld dat in het algemeen meer geavanceerde methoden, zoals de Kalman filter, worden toegepast op dergelijke problemen. Hiervoor zijn detectiemethoden vereist die niet altijd tot de beschikking staan, zoals sonar en beeldherkenning. De Sphero beschikt echter niet over dergelijke detectiemethoden. De keus voor zowel meet- als analysemethode wordt hierdoor beperkt. De wetenschappelijke relevantie van dit onderzoek is dan ook dat, ondanks de beperkingen van de meetapparatuur van een robot, misschien toch een accurate representatie gegeven kan worden van een ruimte. De onderzoeksvraag die hieruit voortkomt is als volgt geformuleerd:

Is een robotbal met collision detectie toereikend om een onbekende ruimte te verkennen?

De hypothese is dat een robotbal in staat is om een relatief grove plattegrond te kunnen maken van een eenvoudige ruimte, zoals een gang. Om de onderzoeksvraag te kunnen beantwoorden zal een Android applicatie geschreven worden die in staat is een Sphero robotbal aan te sturen. Er is gekozen voor deze opzet omdat de SDK van Sphero voor Android applicaties uitgebreider is dan de SDK voor desktopaansturing. De robotbal moet uiteindelijk autonoom door een gang kunnen bewegen, waarbij hij collisions detecteert en hier adequaat op reageert. Na deze verkenningsfase maakt het programma op basis van de gevonden datapunten een representatie van de ruimte. De Sphero robotbal rijdt vervolgens terug naar het punt waar hij is begonnen, zonder tegen de muren te botsen. De verwachting is dat de robotbal in de meeste gevallen in staat is om, wanneer hij in een willekeurige plek in de gang wordt geplaatst, zonder botsingen terug te bewegen naar zijn startpunt.

Methode

Collision detectie

Er is begonnen met het detecteren van collisions met de muur. De Sphero robotbal beschikt over verschillende sensoren, waaronder een acceleratiemeter en een gyroscoop (Sphero Android SDK, 2013). Op basis van deze sensoren zijn er door de fabricant Orbotix al een *DataListener* (waarmee de locatie van de robot bepaald kan worden) en een *CollisionListener* (waarmee collisions gesignaleerd kunnen worden) geïmplementeerd. De DataListener bevat een *LocatorData*-object, welke een x- en y-positie bevat van de robot ten opzichte van zijn beginpunt. De y-as is hierbij gelijk aan een oriëntatie van 0 graden, zie voor een toelichting hierop figuur 1. De CollisionListener detecteert plotselinge veranderingen in snelheid en bepaalt op basis daarvan of er een collision heeft plaats gevonden. Zowel de DataListener als de CollisionListener zijn zogeheten *asynchrone datalisteners*. Dit betekent dat deze twee onderdelen als losse threads draaien en alleen berichten sturen naar de main-thread wanneer zij over nieuwe data beschikken.

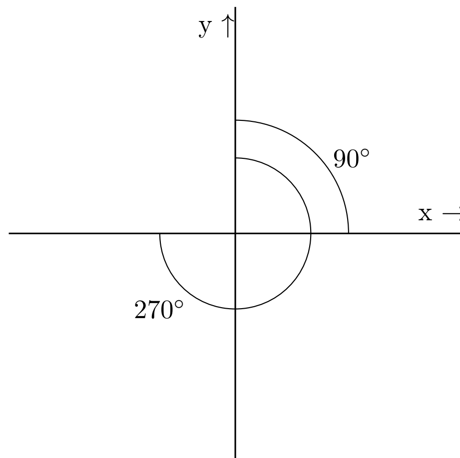


Figure 1: De oriëntatie van de Sphero hangt samen met de tweedimensionale representatie van zijn locatie. Een beweging van 0° staat gelijk aan een beweging richting de y-as en de 90° richting de x-as. De richtingen van de y en x-as hangen af van de calibratie van de robot, die handmatig ingesteld kan worden.

De CollisionListener krijgt bepaalde *threshold values* mee (Sphero Android SDK,), welke aangeven vanaf welke waarden van impact een collision wordt gedetecteerd. In dit onderzoek is gekozen voor relatief hoge threshold values, zodat alleen een zware impact gedetecteerd wordt.

Daarnaast dienen de Data- en CollisionListener aan elkaar gekoppeld te worden. Nieuwe localisatiedata wordt daarom opgeslagen in een globale variabele, *LastLocatorData*. Dergelijke data komt 400 maal per seconde binnen. Wanneer een botsing wordt gedetecteerd, wordt de huidige waarde van de *LastLocatorData* opgehaald en opgeslagen in een *Collision* object. Dit object wordt toegevoegd aan een lijst die de collisions bijhoudt. Per collision is nu dus bekend wat de x- en y-positie zijn waarop deze plaatsvond.

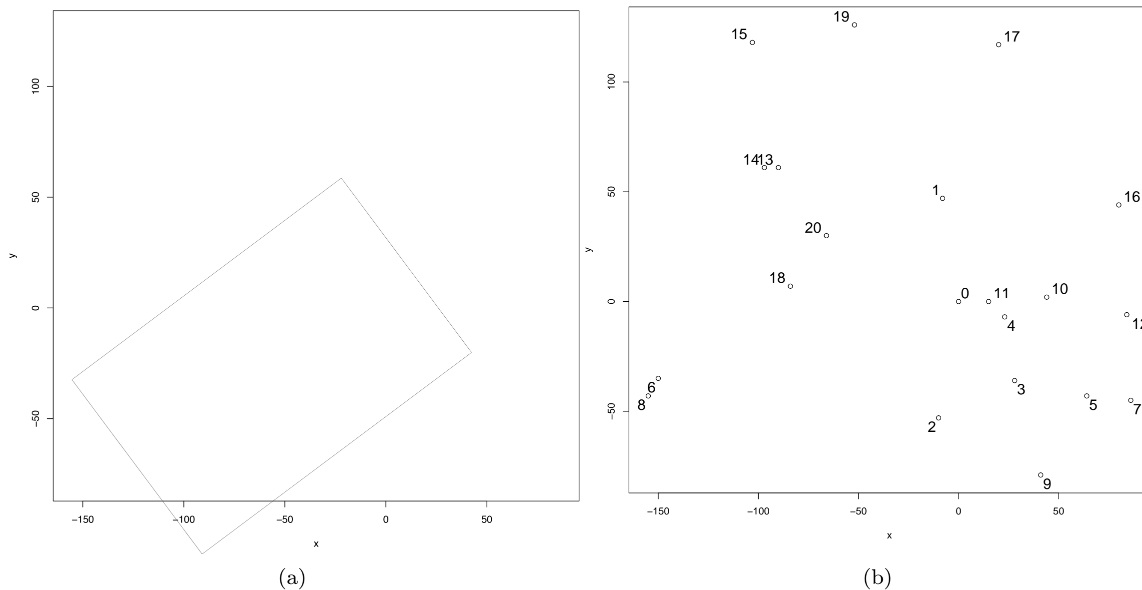
Tijdens het testen bleek dat niet alle collisions worden opgemerkt door de Sphero. Dit zorgt ervoor dat de robot soms stil blijft staan bij een muur. Daarom is er, naast de CollisionListener, een andere vorm van collision detectie geïmplementeerd. Zodra de snelheid van de robot dermate klein is dat hij potentieel stilstaat, wordt er een teller gestart. Wanneer deze lage snelheid langer dan 1 seconde aanhoudt detecteert de robot een stilstand, die wordt verwerkt als een collision. Ook deze data is gekoppeld aan de asynchrone DataListener, zodat de localisatiedata behorende bij de collision bekend is.

Daarnaast bleek ook dat de robot af en toe collisions detecteert wanneer hij wegrijdt. Dit komt door de grote snelheidswisseling die dan plaatsvindt. Dit probleem is opgelost door in de eerste seconde na een beweging tijdelijk de berichten van de CollisionListener te negeren. Het aantal onterecht gedetecteerde collisions is hierdoor afgenomen.

Autonoom rijden

De volgende stap is het autonoom laten rijden van de Sphero. Bij een eerste implementatie bewoog de Sphero op basis van een reflectie met de muur. Uit de data van de CollisionListener kon namelijk bepaald worden in welke mate in een bepaalde richting (x of y) de treshhold values overschreden werden. Op basis hiervan kan de hoek van inval en hoek van uitval bepaald worden (zie Appendix B1). Er is gekozen voor een hoek waarbij de Sphero in de richting van de normaal-as van de muur beweegt. Op deze manier is de kans op een nieuwe collision met de andere muur het grootst. Omdat de muren parallel lopen is de korste afstand tussen beide immers een lijn die parallel loopt met de normaal-as van beide muren. Om te voorkomen dat de robot steeds heen en weer beweegt over dezelfde lijn is een kleine afwijking van 5 graden toegevoegd aan deze hoek (zie Appendix B2 voor de Java implementatie van deze methode).

De Sphero is nu in staat autonoom door een ruimte te bewegen. Echter, door de grote hoekwisselingen wordt de localisatiedata ernstig verstoord. In figuur 2a is te zien wat de Sphero had moeten detecteren bij een van de testrondes. In figuur 2b staan de collisiepunten die de Sphero gedetecteerd heeft. Uit deze datapunten is de beoogde rechthoek nauwelijks te ontdekken. Hieruit blijkt dat de eerste techniek dus niet geschikt is.



(a) (b)

Figure 2: Wat Sphero had moeten zien (links) en wat Sphero detecteerde (rechts) met de eerste methode voor autonome beweging

Er is uiteindelijk voor een methode gekozen waarbij de Sphero in een standaardpatroon heen en weer beweegt door de gang. Eerst beweegt Sphero in een hoek van 90 graden (de linkerbeweging), tot de eerste collision. Vervolgens draait hij om en beweegt in een richting (de rechterbeweging) die over het algemeen

tussen de 275-300 graden zal liggen. Wanneer vervolgens weer een collision wordt gedetecteerd, beweegt de robot weer terug in een hoek van 90 graden. Er ontstaat dan een patroon zoals te zien is in figuur 3a. Er is een handige gebruikersinterface geïmplementeerd voor deze methode. De gebruiker kan hierbij aangeven hoeveel collisions de robot in totaal moet maken en wat de hoek van de linkerbeweging moet zijn. De localisatiepunten die nu gemeten worden door de Sphero zijn liggen meer in lijn met wat men zou verwachten. In figuur 3b is te zien dat de nieuwe datapunten duidelijk twee parallele lijnen, de twee muren van de gang, beslaan.

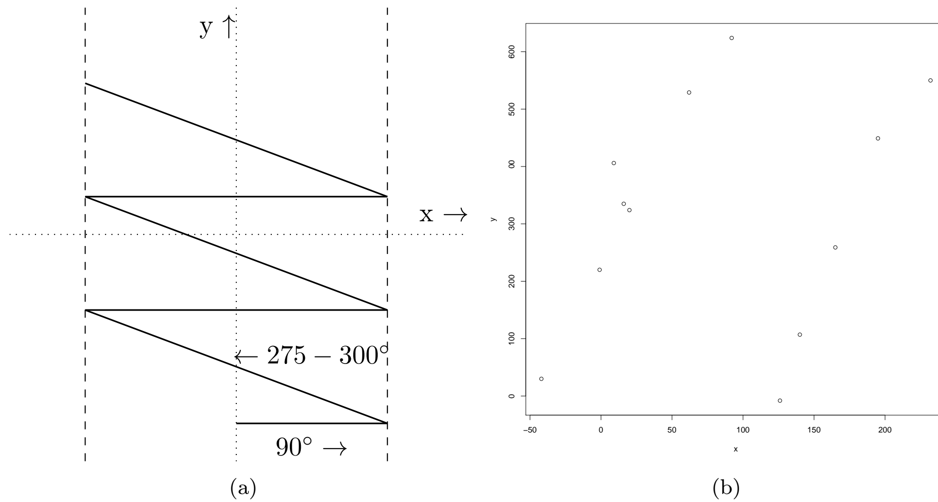
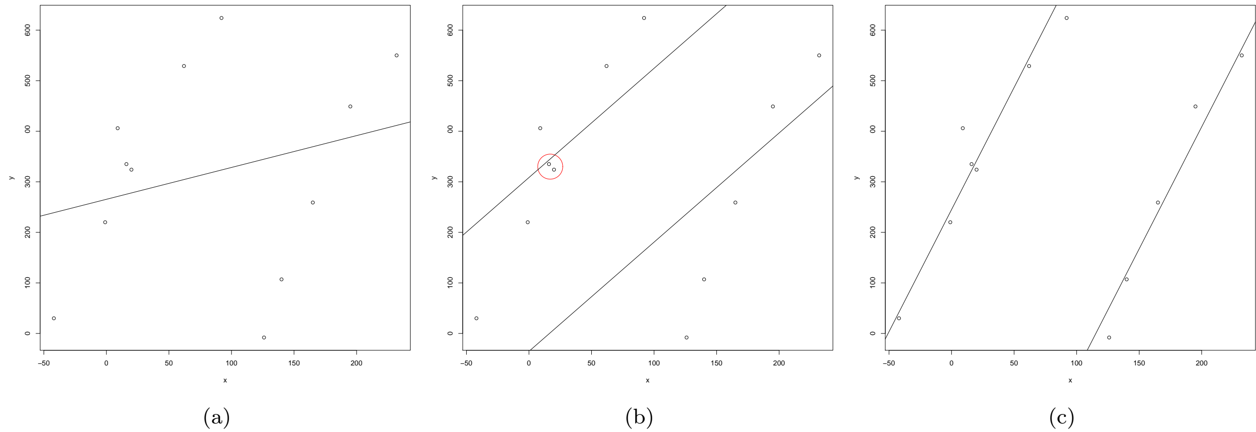


Figure 3: Waneer de Sphero de route rijdt zoals aangegeven in figuur a, resulteert dit in meetpunten zoals in figuur b

Data fitting

De gevonden datapunten dienen nu geanalyseerd te worden. Hiervoor zijn verschillende methoden uitgeprobeerd, allen gebaseerd op de *sum of least squares data fit* (Bretscher, 2009). In eerste instantie werd een analyse gedaan van alle datapunten, zoals te zien in figuur 4a. Het idee was om de data vervolgens te splitsen op basis van deze lijn en weer een least square fit op de nieuwe sets toe te passen. Uit figuur 4b blijkt echter duidelijk dat deze methode niet werkt. De lijn klopt wanneer men puur naar uitvoering van de sum of least squares kijkt, maar is geen scheidingslijn tussen de datapunten van beide muren.

De data kan echter ook op een andere manier gesplitst worden. De datapunten zijn namelijk zeer gestructureerd: men weet welke coördinaten van de linkermuur en welke van de rechtermuur komen. Met die informatie is het mogelijk de data te splitsen en op beide datasets een sum of least squares toe te passen. Wanneer men vervolgens ook meeneemt dat beide muren parallel lopen aan elkaar, leidt dit tot de volgende vergelijking (Fitting two parallel lines, 2013):



(a) (b) (c)

Figure 4: De datafit van de punten op basis van een enkele least square (a), een parallelle least square (b) en een parallelle least square met filtering van ongeloofwaardige datapunten (c), die rood omcirkeld zijn in figuur 4b

$$\begin{bmatrix} 1 & 0 & x_{A1} \\ 1 & 0 & ... \\ 1 & 0 & x_{An} \\ 0 & 1 & x_{B1} \\ 0 & 1 & ... \\ 0 & 1 & x_{Bn} \end{bmatrix} \begin{bmatrix} c1 \\ c2 \\ rc \end{bmatrix} = \begin{bmatrix} y_{A1} \\ ... \\ y_{An} \\ y_{B1} \\ ... \\ y_{Bn} \end{bmatrix}$$

Oplossen van deze vergelijking met behulp van sum of least squares resulteert in lijnen zoals in figuur 5b. Deze lijnen kloppen al meer met de daadwerkelijke locatie van de muren, maar heeft nog steeds een kleine afwijking. Deze wordt veroorzaakt door de rood omcirkelde datapunten. Deze metingen zijn onstaan doordat de robot, ondanks de pauze van 1 seconde, alsnog zijn eigen versnelling voor een collision aanzag. Door datapunten die minder dan een halve meter van elkaar afliggen uit de dataset te filteren, ontstaan de lijnen zoals in figuur 5c. Dit lijkt al veel meer op wat men intuïtief zou verwachten. Deze analysemethode is daarom geïmplementeerd in de applicatie. Dit is gedaan met behulp met het Jama (Jama Library, 2013), een Java package voor het bewerken van matrices.

Veilig terugkeren

De laatste stap is om de Sphero veilig terug te laten rijden naar het begin van de gang. Bij het terugrijden dient de bal eerst naar het midden van de hal te rijden. Dit gebeurt door het y-coördinaat van de bal op te vragen en deze in te vullen in de lijnvergelijkingen die verkregen zijn bij de datanalyse. Door deze twee waarden te middelen kan de x-coördinaat van het midden van de gang gevonden worden die hoort bij de huidige y-positie van de bal. Op basis van zijn huidige x-coördinaat rijdt de bal vervolgens naar links of naar rechts totdat het midden van de hal bereikt is.

Vervolgens dient de bal door het midden van de gang te rijden. Dit kan door in dezelfde richting te rijden als de richtingscoëfficiënt a van de lijnvergelijkingen aangeven. Deze hoek (θ) kan berekend worden met de volgende formule (zie Appendix C voor de afleiding van deze formule):

$$\theta = 180 + 1/(atan(a))$$

Wanneer de y-coördinaat van de robotbal vervolgens kleiner is dan 0, is de robot ongeveer terug waar hij begonnen is en moet een stopcommando verstuurd worden. De sequentie is dan voltooid.

In de huidige implementatie eindigt de robot zijn op basis van de y-coördinaten van zijn beginpunt. Mooier zou zijn dat de robot zelf ook het begin en einde van een gang kan detecteren. Hiervoor is een theorie opgesteld, maar deze is nog niet daadwerkelijk geïmplementeerd. De theorie stelt een lijn door alle punten die haaks staan op de vector van de hal ten opzichte van een punt met bijvoorbeeld de hoogste of laagste y, de lijn door die punten de afsluiting van de hal weergeeft. In Appendix D is een uitwerking te vinden van deze theorie.

Resultaten

Uiteindelijk bleek Sphero in staat te zijn twee lijnvergelijkingen op te stellen die twee muren representeren. Op basis van deze muren kan de robot de juiste hoek berekenen waarin hij terug moet rijden. In figuur 5a tot en met 5c staan voorbeelden van datafits die de Sphero robot heeft gemaakt. Bij ieder van deze metingen begon de Sphero vanuit een andere calibratie.

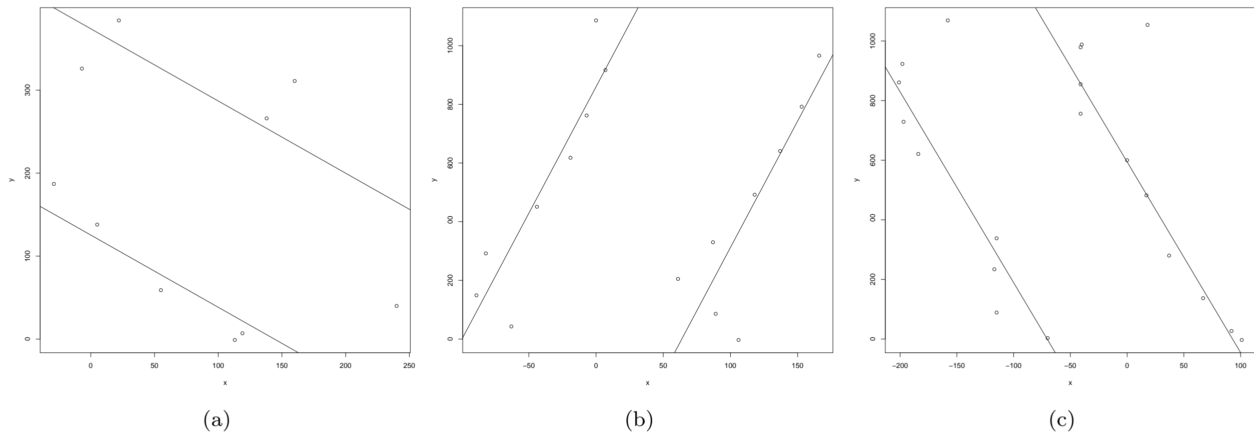


Figure 5: De datafit van drie testrondes van de Sphero bal door een gang, ieder met verschillende calibratie van de Sphero bij het beginpunt

Na het doen van enkele tests bleek dat Sphero in 9 van 10 gevallen in staat was in de juiste richting terug te rijden. Hoewel hij af en toe nog steeds onterecht collisions detecteert bij het wegrijden, lijkt de filter goed genoeg te werken om deze datapunten niet mee te nemen in de berekeningen. Het zoeken van het midden van de kamer lukte slechts in 6 van 10 gevallen. Dit is waarschijnlijk te wijten aan het feit dat Sphero niet direct reageert op het krijgen van een stopsignaal, waardoor hij iets te ver doorrolt. Daardoor eindigt hij niet precies in het midden van de gang.

De uiteindelijke Java implementatie van de Android applicatie die de Sphero aan kan sturen is terug te vinden in Appendix E.

Conclusie en discussie

Uit de resultaten van het onderzoek blijkt dat een robotische bal inderdaad in staat is om tot op zekere hoogte een plattegrond te maken van een gang. Het gebruik van collisions als meetmethode blijkt toereikend genoeg voor het detecteren van een dergelijke ruimte. De robot is in 9 van de 10 gevallen in staat een accurate richting te geven van de gang. Het is echter niet gelukt, zoals gesteld werd in de hypothese, om dit vanuit een willekeurig punt in de ruimte te doen. De robot beweegt namelijk in principe maar in één richting de gang in en dient daarom aan het begin van de gang geplaatst te worden. De oriëntatie van de robotbal mag overigens wel scheef zijn ten op zichte van de gang om hem nog steeds goed te laten functioneren. Dit blijkt doordat ook met verschillende calibraties een correcte analyse werd gedaan.

Er zijn gedurende dit onderzoek een aantal beperkingen geconstateerd. Allereerst bleek dat de robot niet in staat is om ruimtes te mappen met een ingewikkeldere structuur dan een gang. Deze moeilijkheid ligt ten eerste in het analyseren van de resultaten. Een least squares data fit van de tweede orde kan alleen rechte lijnen door data trekken en is dus niet in staat om bijvoorbeeld vierkante ruimtes te detecteren. Andere methoden, zoals bijvoorbeeld de in Appendix D uitgewerkte theorie voor het bepalen van het einde van de gang, zijn daarvoor een vereiste. Daarnaast is de betrouwbaarheid van de localisatie een struikelpunt. De oplossing voor dit probleem, een gestructureerde bewegingssequentie, is niet geschikt voor het verkennen van alle soorten ruimtes. Neem bijvoorbeeld figuur 6. Wanneer de robot in vlak A begint met rijden met onze techniek, zal vlak C nooit bereikt worden.

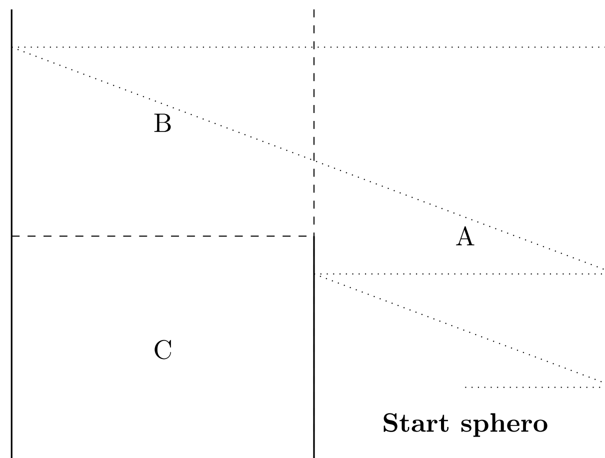


Figure 6: De muren worden gerepresenteerd als dichte lijnen, de vlakken worden van elkaar onderscheiden door gestreepte lijnen. De beweging van de Sphero zijn de gestippelde lijnen. Met de huidige implementatie kan de Sphero een mapping doen van gebied A en B, maar zal hij nooit gebied C bereiken

De huidige implementatie zou op verschillende manieren verbeterd kunnen worden. Allereerst kan de applicatie in gebruikersgemak verhoogd worden door het toevoegen van een calibratiefunctionaliteit. Hiermee zou vooraf de oriëntatie van de Sphero bepaald kunnen worden. Daarnaast zou ook de mapping verbeterd kunnen worden. Denk bijvoorbeeld aan de implementatie van de theorie voor het detecteren van het einde van een gang. Ook zou het interessant kunnen zijn om de datapunten op andere intelligente manieren op te delen en te analyseren, waarbij misschien ook bochten en hoeken in ruimtes gevonden kunnen worden. De robot zou dan eventueel ook in staat zijn objecten in een ruimte, zoals een stoel, te detecteren en deze later te ontwijken. Vervolgonderzoek moet dit uitwijzen.

Referenties

Bretscher, O. (2009) *Linear algebra with applications (4th ed)*. Upper Saddle River, NJ: Prentice Hall.

Fitting two parallel lines. (n.d.). In MatlabKurs. Retrieved June 27, 2013, from *http://people.inf.ethz.ch/arbenz/MatlabKurs/node86.html*

Jama Library. (n.d.). MathWorks. Retrieved June 27, 2013, from http://math.nist.gov/javanumerics/jama/

Sphero Android SDK. (n.d.). Orbotix. Retrieved June 22, 2013, from https://github.com/orbotix/Sphero-Android-SDK

S, Thrun. (2005) *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents series.* Wolfram Burgard, Dieter Fox.

Appendix A: Gebruikt materiaal

Hardware

1. Sphero S002IN
2. Macbook Pro 2010 met OS x 10.8.2
3. HTC desire S met Android 2.2.3
4. HTC desire C met Android 4.2.2

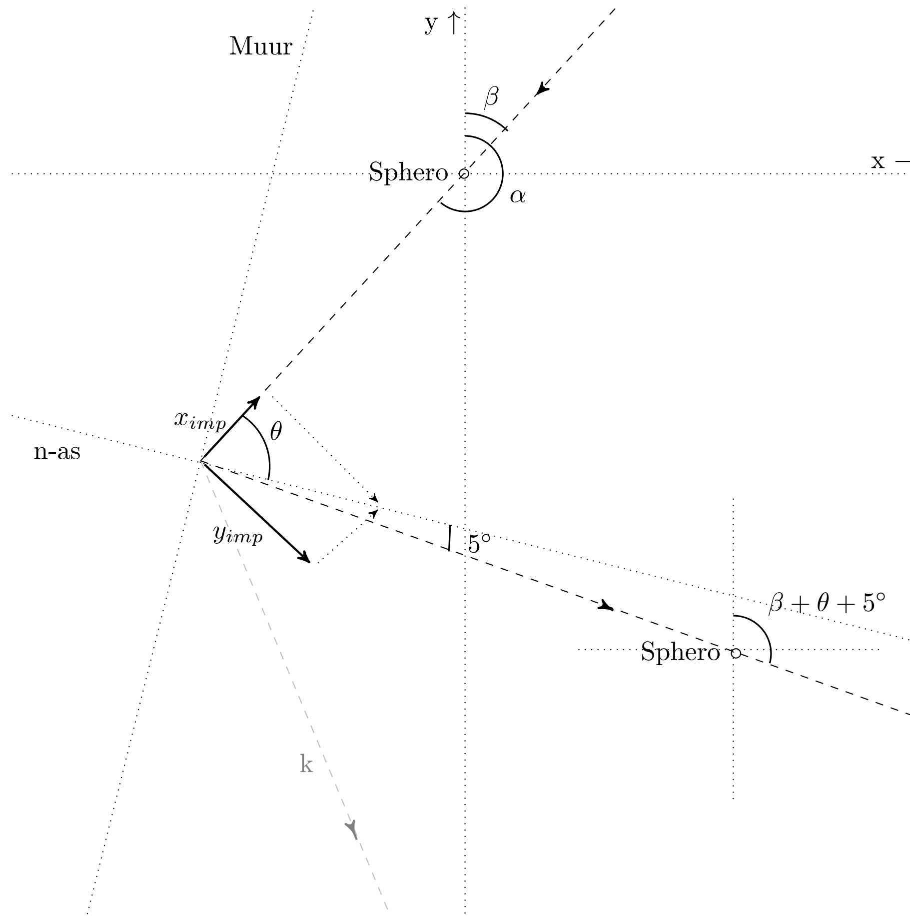
Software

1. Eclipse (Android Development Tools Edition)
2. Java compiler 1.6

Gebruikte libraries

1. RobotLibrary.jar (Sphero Android SDK)
   Te downloaden op: *https://github.com/orbotix/Sphero-Android-SDK*
2. Jama.jar (Java Matrix 1.0.3)
   Te downloaden op: *http://math.nist.gov/javanumerics/jama/*

Appendix B1: Berekening hoek voor reflectie tegen muur



Om de hoek van weerkaatsing te bepalen, dient eerst de hoek van inval, θ, bepaald te worden. Dit is de hoek van de beweegrichting ten opzichte van de normaal-as van de muur, de lijn die loodrecht op de muur staat. Deze hoek kan berekend worden door gebruik te maken van de x- en y-impact. De formule voor θ is dan:

$$\theta = atan(x_{impact}/y_{impact})$$

Om een natuurlijke weerkaatsing vanuit beweegrichting α te simuleren moet de hoek van weerkaatsing ten opzichte van de richting waar Sphero vandaan kwam gelijk zijn aan:

$$\text{Hoek van weerkaatsing} = 2*\theta$$

Deze weerkaatsing is in de figuur weergeven als lijn k. Omdat de Sphero uitgaat van zijn eigen tweedimensionale stelsel voor het bepalen van de richting waarin hij moet gaan, kan $2*\theta$ niet direct worden toegepast. Uit de figuur blijkt dat de hoek van terugval gelijk is aan $\theta*2 + \beta$, waarbij voor β geldt:

$$\beta = \alpha - 180°$$

Er is gekozen om Sphero niet met deze hoek te laten weerkaatsen. Voor het detecteren van een gang is het namelijk wenselijker dat de Sphero met de normaal-as meebeweegt. De normaalas van beide muren zijn namelijk gelijk, en een beweging over de normaal-as is dan de kortste afstand tussen beide muren. Sphero maakt op deze manier zo snel mogelijk weer een collision. Om te voorkomen dat de Sphero heen en weer

blijft kaatsen zonder vooruit te komen in de gang, wordt er een kleine afwijking toegevoegd van 5°. De uiteindelijke nieuwe beweegrichting van de Sphero is dan als volgt:

$$\alpha_n ieuw = \beta + \theta + 5°$$

Appendix B2: Code snippet berekening

De uiteindelijke hoekberekening is als volgt in Java geïmplementeerd:

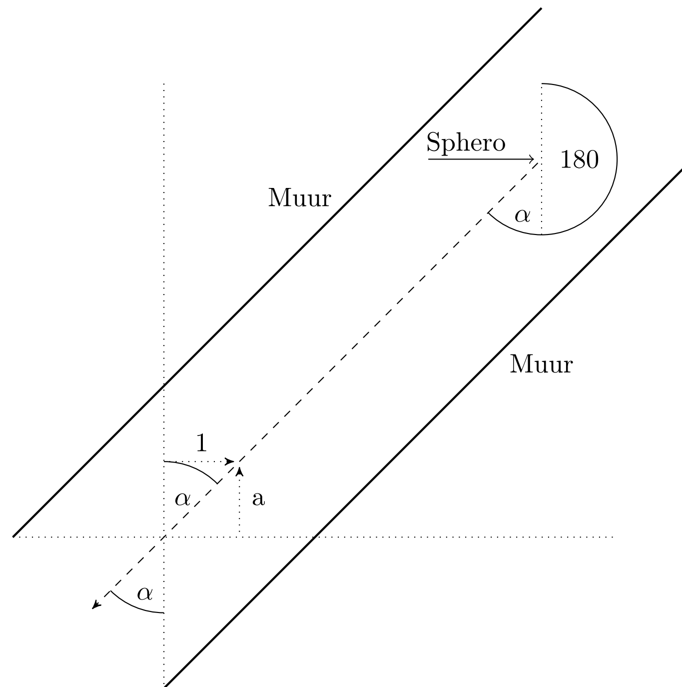
```
// Retrieve collision power
CollisionPower power = collisionData.getImpactPower();

// Calculate new angle to drive to (lastPressed is previous angle of movement)
double theta = Math.toDegrees(Math.atan((double) power.x /(double) power.y));
double newAngle = lastPressed - 180 + theta + 5;

// Replace negative angles by positive angles
if(newAngle < 0) {
        newAngle += 360;
}

RollCommand.sendCommand(mRobot, (int) newAngle, 0.6f);
lastPressed = (int) newAngle;
```

Appendix C: Berekenen hoek van terugkeren



Wanneer de Sphero vanuit het einde van de gang terug moet keren naar zijn beginpunt, moet zijn richting gelijk zijn aan de negatieve richtingscoëfficient van de opgestelde lijnvergelijkingen.

Uit de figuur blijkt dat $\angle\alpha$ gelijk is aan de hoek die de muur maakt met 0-oriëntatie (Z-figuur). Deze hoek hangt af van de richtingscoëfficient a van de lijnvergelijking. Met behulp van de tangens kan a als volgt berekend worden:

$$\angle\alpha = atan(1/a)$$

Omdat de Sphero in tegengestelde richting moet rijden, moet aan deze hoek nog 180° worden toegevoegd. Dit blijkt ook uit de figuur. De uiteindelijke draaiingshoek vanaf de 0-oriëntatie (de y-as) van de Sphero is dus gelijk aan:

$$180° + \angle\alpha$$

Appendix D: Theorie voor bepalen einde van de gang

In de huidige implementatie is de Sphero alleen in staat de zijmuren uit de data te onderscheiden. Interessant zou zijn om ook het einde en het begin van de gang te kunnen detecteren. Hiervoor is de volgende theorie opgesteld.

Ervan uitgaande dat er meerdere collisions worden gevonden tegen de muur die de hal afsluit, kan een functie (zie hieronder) geschreven worden. Deze functie controleert of een gegeven punt haaks op de lijnvergelijking van de de hal staan vanuit een gegeven startpunt. Dit startpunt kan bijvoorbeeld de hoogste y zijn, of, wanneer men gebruik maakt van een marge, een punt dat in de buurt ligt van de hoogste y. Dit laatste kan nuttig zijn omdat de gemeten datapunten een bepaalde foutmarge bevatten en dus niet exact loodrecht op een ander datapunt zullen staan, zelfs al doen zij dit in werkelijkheid wel. Door de punten die op deze manier worden bepaald kan nu de sum of least squares worden toegepast. De lijn die hiermee wordt gevonden zou dan de afsluiting van de hal kunnen representeren.

```
private static boolean inLine(CollisionPoint col1, CollisionPoint col2, double
    c1 ){
        boolean bool = false;

        // Original function: tests if point are exactly on the same line
        // if( (col1.y-col2.y) == (col1.x-col2.x)*(-c1) ){

        // Altered function: tests if points are on the same line within a
            margin of 10% on either side.
        if( ( (col1.y-col2.y) < (col1.x-col2.x)*(-c1*0.9) ) && ( (col1.y-col2.
            y) > (col1.x-col2.x)*(-c1*1.1) ) ){
                bool = true;
        }

        return bool;
}
```

Appendix E: De code van de applicatie

```java
/**
 * CorridorMappingActivity.java
 *
 * Application for detecting a corridor with a Sphero robotic ball
 *
 * @author Lotte Weerts
 */

import android.app.Activity;
import android.os.Bundle;
import android.os.Handler;
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;
import orbotix.robot.base.*;
import orbotix.robot.sensor.DeviceSensorsData;
import orbotix.robot.sensor.LocatorData;
import orbotix.view.connection.SpheroConnectionView;
import orbotix.view.connection.SpheroConnectionView.OnRobotConnectionEventListener;
import java.util.List;
import java.util.ArrayList;
import Jama.Matrix;

public class CorridorMappingActivity extends Activity {
  /** Angle of the right move in the sequence (set by user) */
  int RETURN ANGLE = 275;

  /** Amount of collisions before the calculation of the mapping starts
   * (set by user) */
  int MAX COLLISION = 10;

  /** True if calculation completed and moving to middle of corridor */
  boolean toMiddle = false;

  /** True if moved to middle of corridor and going back to start position */
  boolean toEnd = false;

  /** Center of corridor, set after calculations */
  double xCenter = 0;

  /** Angle of corridor in terms of Sphero’s orientation, set after calculations */
  int finalAngle = 0;

  /** Contains last detected location */
  private LocatorData lastLocatorData;

  /** Counter for the stillstand collision detection */
  int velocityCount = 0;

  /** Determines which angle was lastly moved to */
  private int lastPressed = -1;

  /** Contains all the detected collisions */
  private ArrayList<CollisionPoint> collisionList = new ArrayList<CollisionPoint>();

  /** True if collision detection should be active */
  boolean activeColDec = true;

  /**
   * Robot to from which we are streaming
   */
  private Robot mRobot = null;

  /**
   * The Sphero Connection View
   */
  private SpheroConnectionView mSpheroConnectionView;

  /**
   * Handler of android application
   */
  private Handler mHandler = new Handler();

  /**
   * Called when the activity is first created.
   */
  @Override
```

```java
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  findViewById(R.id.back layout).requestFocus();
  mSpheroConnectionView = (SpheroConnectionView) findViewById(R.id.sphero connection view);

  // Set the connection event listener to connect with Sphero
  mSpheroConnectionView
      .setOnRobotConnectionEventListener(new OnRobotConnectionEventListener() {
        // If the user clicked a Sphero and it failed to connect
        @Override
        public void onRobotConnectionFailed(Robot robot) {
        }

        // If there are no Spheros paired to this device
        @Override
        public void onNonePaired() {
        }

        // The user clicked a Sphero and it successfully paired.
        @Override
        public void onRobotConnected(Robot robot) {
          mRobot = robot;
          mSpheroConnectionView.setVisibility(View.GONE);

          // This delay post is to give the connection time to be created
          mHandler.postDelayed(new Runnable() {
            @Override
            public void run() {

              // Start streaming Locator values
              requestDataStreaming();

              // Set the AsyncDataListener that will process each response.
              DeviceMessenger.getInstance().addAsyncDataListener(mRobot,
                      mDataListener);
              // Set collision listener
              DeviceMessenger.getInstance().addAsyncDataListener(mRobot,
                      mCollisionListener);
              // Set paramaters for collision detection
              ConfigureCollisionDetectionCommand.sendCommand(mRobot,
                      ConfigureCollisionDetectionCommand.DEFAULT DETECTION METHOD,
                      200, 0, 100, 0, 1000);
            }
          }, 1000);
        }

        @Override
        public void onBluetoothNotEnabled() {
          Toast.makeText(LocatorActivity.this,
              "Bluetooth_Not_Enabled", Toast.LENGTH LONG)
              .show();
        }
      });
}

/**
 * Called when the user comes back to this app
 */
@Override
protected void onResume() {
  super.onResume();
  // Refresh list of Spheros
  mSpheroConnectionView.showSpheros();
}

/**
 * Called when the user presses the back or home button
 */
@Override
protected void onPause() {
  super.onPause();
  // Remove async locater listener
  DeviceMessenger.getInstance().removeAsyncDataListener(mRobot,
      mDataListener);

  // Remove async collision listener
  DeviceMessenger.getInstance().removeAsyncDataListener(mRobot,
      mCollisionListener);
```

```
  // Disconnect Robot properly
  RobotProvider.getDefaultProvider().disconnectControlledRobots();
}

/**
 * AsyncDataListener that is used to update the last location variable
 *
 */
private DeviceMessenger.AsyncDataListener mDataListener =
                                      new DeviceMessenger.AsyncDataListener() {
  @Override
  public void onDataReceived(DeviceAsyncData data) {

    if (data instanceof DeviceSensorsAsyncData) {

      // Get the frames in the response
      List<DeviceSensorsData> data list = ((DeviceSensorsAsyncData) data)
          .getAsyncData();
      if (data list != null) {

        // Iterate over each frame and retrieve locator data
        for (DeviceSensorsData datum : data list) {
          LocatorData locatorData = datum.getLocatorData();
          if (locatorData != null) {

            // The locator data must be available to the whole class
            lastLocatorData = locatorData;

            // Detect if ball is too long on same position: a collision
            if (detectStop()) {
              RGBLEDOutputCommand.sendCommand(mRobot, 0, 255, 0);
              collisionHandler(locatorData);
            }

            // Detect if middle of corridor has been reached after calculations
            if (toMiddle
                && locatorData.getPositionX() > xCenter * 0.9
                && locatorData.getPositionX() < xCenter * 1.1) {
              toMiddle = false;
              toEnd = true;

              // Move in direction of parallel lines
              RollCommand.sendStop(mRobot);
              RollCommand.sendCommand(mRobot, finalAngle, 0.6f);
              RGBLEDOutputCommand.sendCommand(mRobot, 0, 100, 170);

            }

            // Detect if start has been reached while rolling through corridor
            if (toEnd && locatorData.getPositionY() < 0) {
              toEnd = false;
              RollCommand.sendStop(mRobot);
              RGBLEDOutputCommand.sendCommand(mRobot, 100, 0,
                  170);
            }
          }
        }
      }
    }
  }
};

/**
 * Set up data streaming for retrieving locator data
 */
private void requestDataStreaming() {

  if (mRobot == null)
    return;

  // Set up a bitmask containing the sensor information we want to stream
  final long mask = SetDataStreamingCommand.DATA STREAMING MASK LOCATOR ALL;

  // 400 hz per second
  final int divisor = 1;

  // Number of frames in each response
  final int packet frames = 20;
```

```
  // Number of async data before it stops is infinite
  final int response count = 0;

  // Send this command to Sphero to start streaming
  SetDataStreamingCommand.sendCommand(mRobot, divisor, packet frames,
      mask, response count);
}

/**
 * Detects if a collision has occurred and changes the color of the robot before
 * starting the collisionHandler
 */
private final AsyncDataListener mCollisionListener = new AsyncDataListener() {

  public void onDataReceived(DeviceAsyncData asyncData) {
    if (asyncData instanceof CollisionDetectedAsyncData && activeColDec) {

      LocatorData location = lastLocatorData;
      RGBLEDOutputCommand.sendCommand(mRobot, 255, 0, 0);
      collisionHandler(location);

    }
  }
};

/**
 * Used to pause the collision listener by temporary changing the activeColDec
 * variable
 *
 * @param time in milliseconds no collisions should be detected
 */
private void pauseCollisionDetection(int time) {
  activeColDec = false;

  mHandler.postDelayed(new Runnable() {
    @Override
    public void run() {
      activeColDec = true;
    }
  }, time);

}

/**
 * Called after a collision has been detected and saves the proper
 * information in the collisionList.
 *
 * @param lastLocatorData
 */
private void collisionHandler(LocatorData lastLocatorData) {
  // Add collision to collisionlist
  CollisionPoint newCollision = new CollisionPoint(
      lastLocatorData.getPositionX(), lastLocatorData.getPositionY());
  collisionList.add(newCollision);

  // If making left move, now go to right
  if (lastPressed == 90) {
    pauseCollisionDetection(1000);
    RollCommand.sendCommand(mRobot, RETURN ANGLE, 0.6f);
    lastPressed = RETURN ANGLE;

  // If making right move, now go to left
  } else if (lastPressed == RETURN ANGLE) {
    pauseCollisionDetection(1000);
    RollCommand.sendCommand(mRobot, 90, 0.6f);
    lastPressed = 90;
  }

  // Change color back to default
  mHandler.postDelayed(new Runnable() {
    @Override
    public void run() {
      RGBLEDOutputCommand.sendCommand(mRobot, 255, 255, 255);
    }
  }, 1000);

  // Only handle max bounces and then start calculations
  if (collisionList.size() == MAX COLLISION) {
    RollCommand.sendStop(mRobot);
    RGBLEDOutputCommand.sendCommand(mRobot, 170, 0, 100);
```

```
    lastPressed = -1;
    double b1, b2, a;

    // Filter collisions that are too close (< 50 cm) to each other
    for (int i = 0; i < (collisionList.size() - 1); i = i + 2) {
      if (Math.abs(collisionList.get(i).x - collisionList.get(i + 1).x) < 50) {
        collisionList.remove(i + 1);
        collisionList.remove(i);
        i = i - 2;
      }
    }

    // Calculate two lines parallel to the corridor, are of the form ax + b1 and
    // ax + b2
    try {
      Matrix c = calculateCentralLine(collisionList);
      b1 = c.get(0, 0);
      b2 = c.get(1, 0);
      a = c.get(2, 0);
      finalAngle = 180 + (int) Math.toDegrees(Math.atan(1 / a));

    // Return defaults when singular matrix is detected, occurs when all
      // points are on an exact vertical line
    } catch (RuntimeException e) {
      b1 = 0;
      b2 = 0;
      a = 0;
      finalAngle = 180;
    }

    // Calculate center x coordinate and start moving to that point
    xCenter = (2 * lastLocatorData.getPositionY() - b1 - b2) / a;
    toMiddle = true;

    if (lastLocatorData.getPositionX() > xCenter) {
      RollCommand.sendCommand(mRobot, 270, 0.6f);
    } else {
      RollCommand.sendCommand(mRobot, 90, 0.6f);
    }
  }
}

/**
 * Detects if velocity has been lower than 20 for more than 1 second
 * @return true if standing still, false otherwise
 */
private boolean detectStop() {
  double velocity = Math
      .sqrt(Math.pow(lastLocatorData.getVelocity().x, 2)
          + Math.pow(lastLocatorData.getVelocity().y, 2));

  if (velocity > -25 && velocity < 25 && lastPressed >= 0) {
    velocityCount++;
  } else {
    velocityCount = 0;
  }

  if (velocityCount == 400) {
    return true;
  }

  return false;
}

/**
 * When the user clicks the configure button, it calls this function
 *
 * @param v
 */
public void configurePressed(View v) {
  collisionList = new ArrayList<CollisionPoint>();
  if (mRobot == null)
    return;

  // Reset the configuration of the robot to zero
  int newX = 0;
  int newY = 0;
  int newYaw = 0;

  // Locator grid should be rotated with calibration
```

```
  int flag = 1;

  ConfigureLocatorCommand.sendCommand(mRobot, flag, newX, newY, newYaw);

  // Parse information from text fields
  try {
    MAX COLLISION = Integer.parseInt(((EditText) findViewById(R.id.edit count))
            .getText().toString());
  } catch (NumberFormatException e) {}

  try {
    RETURN ANGLE = Integer.parseInt(((EditText) findViewById(R.id.edit angle))
            .getText().toString());
  } catch (NumberFormatException e) {}
}

/**
 * Move Sphero to 0 degrees
 *
 *  @param v
 */
public void upPressed(View v) {
  lastPressed = 0;
  RollCommand.sendCommand(mRobot, 0, 0.6f);
}

/**
 * Move sphero to 90 degrees (starts sequence)
 *
 * @param v
 */
public void rightPressed(View v) {
  // Make sure no collision is detected at first second
  pauseCollisionDetection(1000);
  lastPressed = 90;
  RollCommand.sendCommand(mRobot, 90, 0.6f);
}

/**
 * Move Sphero to final angle (used if detection of middle failed)
 *
 * @param v
 */
public void leftPressed(View v) {
  lastPressed = finalAngle;
  toMiddle = false;
  RollCommand.sendCommand(mRobot, finalAngle, 0.6f);
}

/**
 * Make Sphero stop and reset program
 *
 * @param v
 */
public void stopPressed(View v) {
  RollCommand.sendStop(mRobot);
  collisionList = new ArrayList<CollisionPoint>();
  lastPressed = collisionList.size();
}

/**
 * Retrieves list of x coordinates from collisionList
 *
 * @param collisionList
 * @return list of x coords
 */
private static double[] getX(ArrayList<CollisionPoint> collisionList) {
  double[] xcoords = new double[collisionList.size()];

  for (int i = 0; i < (collisionList.size()); i++) {
    xcoords[i] = collisionList.get(i).x;
  }

  return xcoords;
}

/**
 * Retrieves list of y coordinates from collisionList
 *
 * @param collisionList
```

```
 * @return list of x coords
 */
private static double[] getY(ArrayList<CollisionPoint> collisionList) {
  double[] ycoords = new double[collisionList.size()];

  for (int i = 0; i < (collisionList.size()); i++) {
    ycoords[i] = collisionList.get(i).y;
  }

  return ycoords;
}

/**
 * Calculates parameters for line equation for the walls of the corridor
 * from a filtered collisionList
 *
 * @param collisionList
 * @return parameters for line equation
 */
private static Matrix calculateCentralLine(
    ArrayList<CollisionPoint> collisionList) {
  double[] xcoords = getX(collisionList);
  double[] ycoords = getY(collisionList);

  Matrix A = new Matrix(createA(xcoords));
  Matrix b = new Matrix(createB(ycoords));

  Matrix c = leastSquares(A, b);
  return c;

}

/**
 * Performs least squares on the matrices A and b to calculate
 * c
 *
 * @param A
 * @param b
 * @return result of least squares calculation
 */
private static Matrix leastSquares(Matrix A, Matrix b) {
  Matrix AT = A.transpose();
  Matrix AtA = AT.times(A);
  Matrix AtA1 = AtA.inverse();
  Matrix Atb = AT.times(b);
  Matrix c = AtA1.times(Atb);

  return c;
}

/**
 * Formats the list of x coordinates to a list of doubles representing
 * the A matrix, a 3 by n matrix
 *
 * @param xcoords
 * @return representation of the A matrix
 */
private static double[][] createA(double[] xcoords) {
  double[][] aValues = new double[xcoords.length][3];
  for (int i = 0; i < xcoords.length; i++) {
    if (i % 2 == 1) {
      aValues[i][0] = 1.0;
      aValues[i][1] = 0;
    } else {
      aValues[i][0] = 0;
      aValues[i][1] = 1.0;
    }

    aValues[i][2] = xcoords[i];
  }
  return aValues;
}

/**
 * Formats the list of y coordinates to a list of doubles representing
 * the b matrix, a 1 x n Matrix
 *
 * @param xcoords
 * @return representation of the A matrix
 */
```

```java
  private static double[][] createB(double[] ycoords) {
    double[][] bValues = new double[ycoords.length][1];
    for (int i = 0; i < ycoords.length; i++) {
      bValues[i][0] = ycoords[i];
    }
    return bValues;
  }

  /**
   * Representation of a collision containing its x and y location
   *
   * @author Lotte Weerts
   */
  public class CollisionPoint {
    /** The x and y coordinates of the collision */
    float x, y;

    CollisionPoint(float x, float y) {
      this.x = x;
      this.y = y;
    }

    public String toString() {
      return new String(x + "\t" + y);
    }
  }

}
```