Woord vooraf

Als laatstejaarsstudenten van de opleiding master in de industriële wetenschappen energie automatisering hebben wij gekozen om onze masterproef in het onderzoekscentrum ACRO te voltooien. Dit is gehuisvest in het technologiecentrum te Diepenbeek.

Industriële robots bewegen altijd van configuratie A naar B en moeten in de industrie opnieuw geprogrammeerd worden om een andere beweging te realiseren. Een robot is “blind” en houdt dus geen rekening met zijn omgeving. In academische onderzoeken werden in het verleden planners ontwikkeld die automatisch een pad berekenen voor dit probleem. Deze planners berekenen een pad van A naar B, waarbij er rekening wordt gehouden met obstakels. Dit zou uitstekend toepasbaar zijn in de hedendaagse industrie, want zo hoeft een robot niet telkens opnieuw geprogrammeerd te worden.

In deze masterproef wordt gebruik gemaakt van ROS, dit is een afkorting voor ‘Robot Operating System’. Dit is een robot softwarebibliotheek waarmee autonome robots geprogrammeerd worden.

Tijdens onze masterproef konden we altijd rekenen op een aantal personen. Deze zouden we dan ook zeer graag willen bedanken.

Graag willen we Ir. De Maeyer Jeroen bedanken voor zijn expertise in ROS. Verder willen we ook onze promotor Prof. dr. Ir. Demeester Eric bedanken, op gebied van robotica en zijn algemene begeleiding. Als laatste willen we onze ouders nog bedanken om ons de kans te geven om verder te studeren en het feit dat ze ons altijd zijn blijven steunen.

Inhoudsopgave

Lijst van tabellen

Lijst van figuren

Abstract

Autonome industriële robots worden steeds vaker in een veranderlijke omgeving geplaatst. De bewegingen van een robot houden echter vaak onvoldoende rekening met deze omgeving, wat botsingen kan veroorzaken. Om dit te voorkomen is er nood aan sensoren en algoritmen die in staat zijn om robotbewegingen aan te passen.

Een dergelijk padplanningsalgoritme berekent een traject van begin- tot eindconfiguratie, rekening houdend met de obstakels in zijn omgeving. Deze masterproef, die uitgevoerd werd binnen de onderzoeksgroep ACRO, heeft tot doel de volgende academische padplanners te vergelijken in simulatie: RRTConnect uit OMPL, CHOMP, STOMP, Cartesische interpolatie en TrajOpt.

Als eerste worden, op basis van literatuurstudies, de padplanners theoretisch met elkaar vergeleken. Om de padplanners praktisch te vergelijken worden cases opgesteld voor een verfrobot, de Universal Robot UR5 en de Kuka KR5. De berekende paden voor de robots worden gesimuleerd in ROS. De omgeving voor de verfrobot wordt gescand via een Kinect v2 en wordt toegevoegd met behulp van Octomap, terwijl de omgeving voor de KR5 en UR5-robot manueel worden aangemaakt.

Voor de verfrobottoepassing, waarvoor een lineaire heen- en weergaande pad van de spuitkop vereist is, zijn TrajOpt en Cartesische interpolatie geschikt omdat deze toelaten om constraints te definiëren. De andere planners zijn niet geschikt voor die beweging. Voor de KR5 en de UR5-robot zijn RRTConnect, CHOMP, STOMP en TrajOpt geschikt om point-to-point bewegingen botsingsvrij uit te voeren.

Abstract in English

Autonomous industrial robots are increasingly adopted in changing environments. However the movements of these robots do not necessarily take these changes in the environment into account, which may cause collisions. To prevent this, there is a need to use sensors and algorithms that are capable of adapting the movements so the obstacles are avoided.

A path planning algorithm like that calculates a trajectory from start to end goal while taking the environment into account. This master thesis, which was conducted within the research group ACRO, aims to compare the following academic path planning algorithms in simulation: RRTConnect, which is part of the OMPL-library, CHOMP, STOMP, Cartesian interpolation and TrajOpt.

The first step was to theoretically compare the planners based on their respective academic papers. For the practical comparison three cases were set up: one using a painting-robot, the Universal Robot UR5 and the Kuka KR5. The calculated paths for the robots are simulated in ROS. The environment for the painting-robot was scanned using a Kinect v2 and added by Octomap. The environments for the KR5-robot and UR5-robot were created manually.

The conclusion for the painting-robot is that TrajOpt and Cartesian Interpolation are suitable because they allow for the addition of constraints. The other planners are not usable for linear reciprocating movements. However RRTConnect, CHOMP, STOMP and TrajOpt work fine with the KR5 and the UR5-robot which execute point-to-point movements.

1 Inleiding

1.1 Situering

Deze masterproef vond plaats aan het onderzoekcentrum ACRO van de KULeuven, gelegen in het Technologiecentrum op de universitaire campus te Diepenbeek. Het acroniem ACRO duidt op de expertise van deze onderzoeksgroep, met name: Automatisering, Computervisie en Robotica [1].

ACRO spitst zich momenteel vooral toe op visie & robotica. De combinatie van een visiesysteem met robotica biedt vele nieuwe mogelijkheden. Een bekend voorbeeld van ACRO is de automatische fruitplukmachine waarbij een robot appels plukt op basis van camerabeelden. Machinevisie lost projecten op, waar voordien geen oplossing voor was. Dit bewijst dat machine-visie gecombineerd met robotica succesvol is [1].

In 2016 waren er meer dan 2 miljoen industriële robots geïmplementeerd over de wereld. Deze robots programmeren neemt veel tijd in beslag. Om de programmeertijd van een industriële robot te verminderen zijn betrouwbare padplanningsalgoritmes nodig [2].

1.2 Probleemstelling

Een robot steeds herprogrammeren is een tijdrovend proces. Er moet dus gezocht worden naar alternatieven die minder tijd in beslag nemen. Eén van deze oplossingen is geautomatiseerde padplanning. Deze algoritmes bepalen zelf de beste manier en volgorde om bepaalde acties uit te voeren en zorgen er dus voor dat een programma niet steeds manueel aangepast moet worden voor elke variatie van een bepaalde handeling.

Het gebruik van automatische padplanning kan voor de industrie veel betekenen aangezien het programmeerproces versneld wordt en tevens meer flexibiliteit biedt. Neem hierbij als voorbeeld een lasbedrijf dat een groot aantal verschillende producten maakt met lasrobots, maar steeds in kleine hoeveelheden. Voor elk product moet er dus een ander lasrobotprogramma geschreven worden, met als gevolg dat er ook vaak van programma gewisseld moet worden. Een automatisch padplanningsalgoritme biedt in zo'n situatie een oplossing. De robot kan bijvoorbeeld met visie zelfstandig een pad berekenen om dit vervolgens te lassen.

Optimale padplanning blijkt veelbelovend te zijn op basis van het huidige academische onderzoek. Deze ontwikkelingen zijn beschikbaar in de vorm van open-source software. Maar deze algoritmes worden zelden toegepast in de industrie omdat het gebruik ervan niet vanzelfsprekend is. Dit komt omdat voor elk project/probleem in de industrie vaak een andere oplossing nodig is. De planners kunnen niet goed om met veel variabelen in de omgeving. Om de planners te gebruiken in de industrie moet er visie geïntegreerd worden, om de omgeving in kaart te brengen. Dit is niet evident, want vaak zijn variabelen zoals lichtinval en beeldinterpretatie van belang. De integratie van een padplanner vraagt veel research om toepasbaar te zijn in de hedendaagse industrie.

1.3 Doelstellingen

In het tot nu toe uitgevoerde onderzoek naar padplanners zijn er verschillende algoritmes ontwikkeld. Enkele voorbeelden hiervan zijn: Descartes [2], STOMP [3], CHOMP [4], TrajOpt [5], OMPL [6], etc. Deze zijn echter voornamelijk academisch georiënteerd. Het toepassen in de industrie vraagt nog verdere ontwikkeling. Het doel van deze masterproef is daarom enkele industriële cases te zoeken waarmee deze planners nog problemen ondervinden en deze planners onderling te vergelijken.

Eerst zal er een literatuurstudie van een aantal bestaande algoritmes uitgevoerd worden. Hierna kunnen de algoritmes theoretisch worden vergeleken. Vervolgens worden deze toegepast en vergeleken op cases. Deze worden gemaakt, in simulatie, voor de volgende robots:

- COCO-verfrobot (lineaire beweging),
- Kuka KR5 (PTP-beweging),
- Universal Robot UR5 (PTP-beweging).

De cases zullen bestaan uit een omgeving (scene) met obstakels die de robot moet ontwijken. De gegenereerde bewegingen door de verschillende padplanners worden vervolgens vergeleken.

Voor de verfrobot wordt eerst de omgeving waargenomen met een Kinect2. Hiervoor wordt Octomap gebruikt welke de ingelezen puntenwolk transformeert naar een compacte 3D-voxelkaart (een Octree). Deze wordt dan toegevoegd als object aan de simulatieomgeving. Ten slotte rekenen de planners voor de verfrobot een geschikt botsingsvrij pad uit. Een echte UR5-robot staat ter beschikking in het onderzoekscentrum ACRO. Hierop zullen de padplanners getest worden in real-life [7].

1.4 Methodiek

Om te starten met deze masterproef moet er eerst een literatuurstudie uitgevoerd worden. Een eerste stap in deze studie is het onderzoeken van de kinematica van een robot. Hiervoor zal er gebruik gemaakt worden van het boek Robotics Modelling, Planning and Control. [1] Dit boek biedt ook een introductie naar het padplannen.

Eens de basiskennis van de kinematica van een robot aanwezig is kan er verder gegaan worden met de studie van de verschillende planners. Dit houdt in dat er eerst onderzocht moet worden welke planningsalgoritmes er bestaan. Het lezen van papers over de bestaande planners zal meer inzicht verschaffen over de voor- en nadelen van elk algoritme.

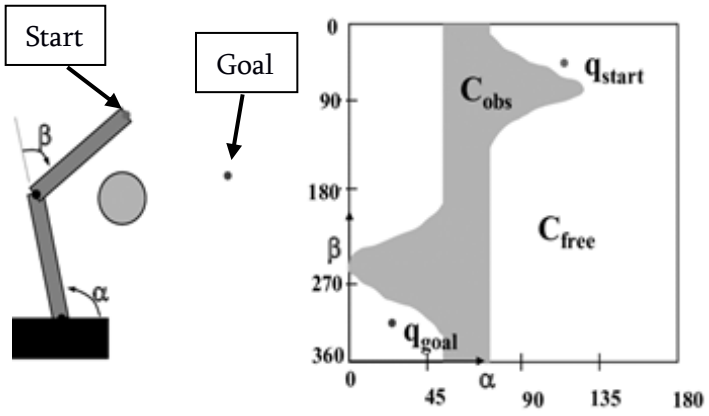
De simulatiesoftware die gebruikt zal worden om alle planners te vergelijken, is “MoveIt!”, een programma dat in combinatie met ROS Kinetic en RVIZ gebruikt wordt. Het leren werken met deze softwarepakketten is dan ook belangrijk om de planner verder te kunnen testen. Om de simulatiesoftware beter te leren kennen wordt er gebruik gemaakt van enkele tutorials die uitgegeven zijn door de community rond “MoveIt!”. Dit simuleren gebeurt in Ubuntu 16.04, een Linux besturingssysteem.

2 Literatuurstudie

2.1 Basisbegrippen bij robotpadplanning

Met behulp van padplanningsalgoritmes kan een pad gegenereerd worden, dit is één van de fundamentele onderzoeksgebieden in de robotica. Vertrekkende van de begin- en eindtoestand van de robot, en in bepaalde gevallen ook van gewenste tussenliggende toestanden zoals de lasnaad, gaat dit algoritme op zoek naar een geschikt botsingsvrij pad. Een robotarm is mechanisch opgebouwd uit "links" die met elkaar verbonden zijn via "joints" en op het einde van de robotarm nog een "end-effector" (vaak een grijper). Het aantal vrijheidsgraden van de robot wordt bepaald door het aantal joints. De configuratieruimte of "C-Space" is de verzameling van alle mogelijke robot configuraties. Dit zijn dus alle mogelijke configuraties die de robot kan bereiken. De dimensie van de "C-Space" is het aantal vrijheidsgraden van de robot [8].

Het padplanningsprobleem is een subonderdeel van het algemene hoofdprobleem onder het thema motion planning. Padplanning kan beschouwd worden als een geometrisch probleem namelijk het vinden van een botsingsvrij pad op basis van de start-configuratie (q_{start}) en de eind-configuratie (q_{goal}). Om de obstakels/hindernissen van de werkruimte concreet te beschrijven bestaat de "C-Space" van de robot uit twee sets van configuraties. De eerste set beschrijft de vrije ruimte (c_{free}), waar de robot in kan bewegen. De tweede set, een obstakelconfiguratieruimte (c_{obs}) definieert de configuraties waarbij de robot bost met obstakels. Met deze opsplitsing wordt het padplanningsprobleem teruggebracht tot het probleem van het vinden van een pad voor een punt-robot tussen de obstakelgebieden. In het voorbeeld in Figuur 1 zijn er twee joints, bijgevolg heeft de C-Space twee dimensies. Het gegenereerde pad hangt af van het planningsalgoritme, de parameters en de probleemstelling. Figuur 1 laat zien dat padplanning van q_{start} naar q_{goal} onmogelijk is omdat q_{start} en q_{goal} niet in hetzelfde deel van c_{free} liggen. Het exact berekenen van de obstakelconfiguratieruimte c_{obs} is erg rekenintensief en in de praktijk niet haalbaar [8].



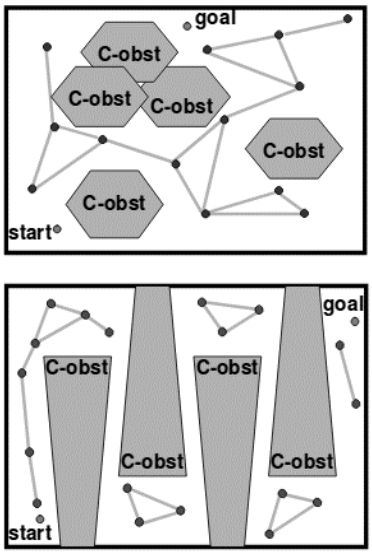
Figuur 1: Voorbeeld van een robot met twee joints en zijn C-Space [9, p. 7].

2.2 Padplanningsalgoritmes

In dit hoofdstuk worden twee verschillende typen van padplanningsalgoritmes besproken. Deze zijn sampling gebaseerde en optimalisatie gebaseerde padplanning.

2.2.1 Sampling gebaseerde padplanning

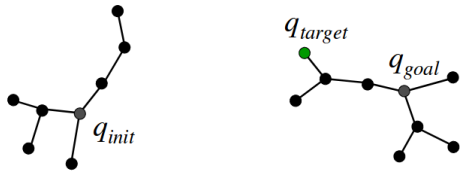
Het creëren van een pad gebeurt in dit geval via het bemonsteren (vandaag "sampling") van de C-Space. Het bemonsteren kan volledig willekeurig of met een zekere systematiek gebeuren. Het algoritme toetst op die manier de C-Space af, creëert een benadering van c_{free} en kan zo efficiënt en snel oplossingen vinden voor moeilijke bewegingen. Rapidly-exploring Random Trees (RRT) is een bekend voorbeeld hiervan en bestaat uit het creëren van een boomstructuur aan mogelijke bewegingen binnen in de vrije C-Space van de robot. Er groeit een boomstructuur vanuit de startconfiguratie door willekeurig samples te kiezen in c_{free} en de boomstructuur in de richting van die samples uit te breiden. Dit gaat voort tot het geselecteerde doel bereikt is. Het RRT-algoritme creëert dus willekeurig een pad naar de target door samples te nemen van zijn omgeving. Figuur 2 toont ook het nadeel van sampling gebaseerde padplanning. Voor sommige problemen vindt het algoritme geen oplossing [10].



Figuur 2: Voorbeeld van padplanning binnen een C-Space met C-Obstakels [11, p. 5].

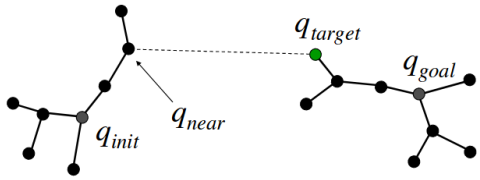
Omdat de berekening van c_{free} met behulp van de samples onvolledig en benaderd is, worden in Figuur 2 nauwe doorgangen niet gevonden en is het onmogelijk om een compleet pad te genereren. Het algoritme komt dus niet altijd tot een oplossing terwijl er misschien wel zijn. Door incrementeel samples toe te voegen aan de boomstructuur, verhoogt wel de kans op het vinden van een pad.

Het RRT-algoritme werkt als volgt. Er zijn verschillende varianten op dit algoritme. In dit hoofdstuk wordt de bi-directionele variant besproken waarbij het de bedoeling is om twee boomstructuren met elkaar te verbinden, één vertrekkend vanuit de startconfiguratie q_{init} en één vanuit de doelconfiguratie q_{goal}. Het RRT-algoritme baseert zich op q_{init}, q_{target} en q_{goal}. Deze robotconfiguraties zijn knooppunten of "nodes" in de boomstructuur in Figuur 3 [9], [10].



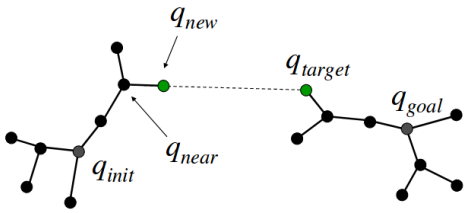
Figuur 3: Begin van een boomstructuur [11, p. 83].

Eerst selecteert RRT een knooppunt q_{target}. Vervolgens zoekt het RRT-algoritme de volgende node q_{near} die zich het dichts bij q_{target} bevindt. Dit wordt aangegeven door q_{near} in Figuur 4.



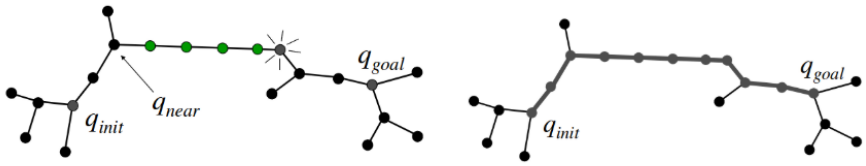
Figuur 4: Creëren van volgende node in de boomstructuur [11, p. 86].

De boomstructuur wordt uitgebreid vanuit q_{near} in de richting van $q_{target,}$ en er wordt een nieuwe node q_{new} gecreëerd, zie Figuur 5.



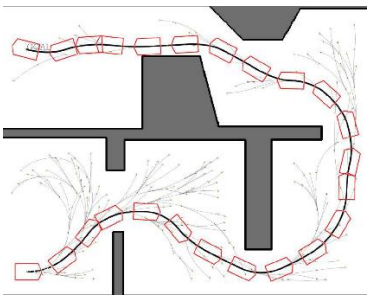
Figuur 5: Creëren van nodes richting 'target' [11, p. 87].

Als dit succesvol is dan kan deze tak verder doorgaan. Als deze een botsing met een object veroorzaakt moet er één stap terug gekeerd worden en moet er opnieuw een vertakking uitgewerkt worden. De nodes worden dan verder opgesteld volgens dit principe. Als laatste stap wordt via de berekende nodes een pad gekozen, weergegeven in Figuur 6. Na deze stap is de berekening afgewerkt [11].



Figuur 6: Volledige gegenereerde boomstructuur [10, p. 91–92].

De robot kan in deze opstelling bewegen van q_{init} naar q_{goal}. Het algoritme controleert echter niet of dat het pad het kortste of efficiëntste is. Dit algoritme wordt vaak gebruikt in complexe omgevingen. Een voorbeeld hiervan is de toepassing op een doolhof, zoals in Figuur 7. Dit is geschikt voor een opbouw van de boomstructuur zoals hierboven vermeld. Het algoritme vindt iteratief een oplossing voor de probleemstelling door samples van zijn omgeving te nemen [10], [12].

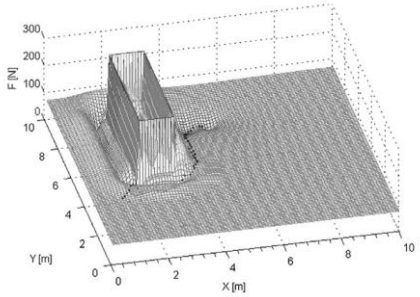


Figuur 7: Voorbeeld van padplanning door het 'groeien' van een boomstructuur [12, p. 7].

2.2.2 Optimalisatie gebaseerde padplanning

De optimalisatie gebaseerde padplanner kan op twee verschillende manieren worden geïmplementeerd. In eerste instantie kan dit algoritme gebruikt worden om een pad te optimaliseren van een ander sampling gebaseerd algoritme. Zo zou deze het pad vloeiender en korter kunnen maken. In tweede instantie kan het algoritme volledig zelfstandig een pad genereren. Hier wordt dan eerst via interpolatie een pad gegenereerd dat waarschijnlijk nog botsingen bevat en geen rekening houdt met bepaalde voorwaarden/constraints. In het vakgebied van robotica wordt de term constraints gebruikt, wat voorwaarden zijn die opgelegd worden door de gebruiker. Het algoritme zal dit gegenereerde pad verder optimaliseren zodat er geen botsingen meer voorkomen met de obstakels in de omgeving en dat er voldaan wordt aan de opgelegde constraints [13].

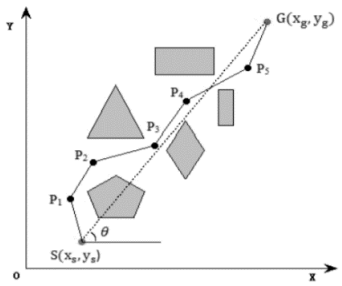
Bij het optimaliseren van een pad wordt er niet alleen getracht om het pad zo vloeiend mogelijk en botsingsvrij te maken maar kan er ook gekeken worden naar de tijd die nodig is om het pad te doorlopen en de daarbij verbruikte energie. De gebruikte methode hiervoor is het toewijzen van een kost aan elk van deze factoren, de grootte van deze kost is afhankelijk van hoeveel belang eraan gehecht wordt. Zo krijgen de configuraties waarbij de robot in botsing is met obstakels een hoge kost toegewezen terwijl de snelheid van het pad of de hoeveelheid energie die nodig is een lage kost krijgt. Deze kosten kunnen gecombineerd worden en kunnen grafisch voorgesteld worden zoals in Figuur 8 [14].



Figuur 8: Grafische weergave van een kostenfunctie [14, p. 29].

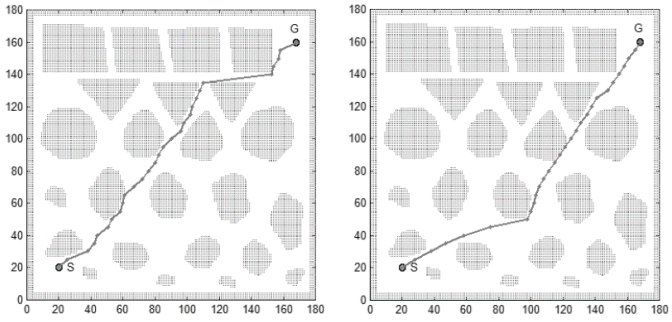
De weergegeven kostenfunctie is bijvoorbeeld van een manipulator. Het ideale pad voor de manipulator is dat pad waarvoor de gemiddelde kost het laagste is. De obstakels, die zich in de omgeving bevinden, krijgen een grote kost toegewezen zodat het algoritme deze probeert te vermijden [15].
Het algoritme zoekt dus een minimum in de kostenfunctie door gebruik te maken van afgeleiden. Wiskundig gezien kan het algoritme de eerste afgeleide berekenen om zo een extremum te herkennen. Vervolgens kan ook de tweede afgeleide berekend worden om een minimum te herkennen. Praktisch gezien is het niet eenvoudig om een lokaal of globaal minimum te onderscheiden. De kostenfunctie heeft vaak vele pieken/dalen en lokale minima. Een wiskundig algoritme zal geen onderscheid kunnen maken tussen een lokaal of globaal minimum. Er zal dus een zoekpatroon nodig zijn om te bepalen waar het globaal minimum gelegen is binnen de kostenfunctie [15].

Een algemeen voorbeeld ter verduidelijking: de robot moet bewegen doorheen zijn omgeving van "Start" naar "Goal". Het algoritme berekent eerst via interpolatie een pad (meestal een rechte lijn) tussen begin- en eindtoestand. Dit pad bevat nog botsingen en is niet geschikt. Door op dit initiële pad een optimalisatie gebaseerd padplanningsalgoritme uit te voeren, creëert de planner wel een botsingsvrij pad. In Figuur 9 wordt weergegeven hoe dit wordt geoptimaliseerd [16].



Figuur 9: Voorbeeld van optimalisatie gebaseerde padplanning [16, p. 4].

In Figuur 10 wordt er een visueel voorbeeld gegeven van mogelijke oplossingen voor een bepaalde probleemstelling. Er bestaan verschillende algoritmes die optimalisatie gebaseerde padplanning gebruiken. Door lineaire interpolatie tussen begin- en eindconfiguratie, een verschillende kostenfunctie die geoptimaliseerd wordt en een verschil in algoritmes kan het gegenereerde pad verschillen. De obstakels worden in beide paden wel vermeden [16].



Figuur 10: Verschil in gegenereerde pad door een verschillend algoritme [16, p. 10].

2.3 Bestudeerde padplanners

In de praktijk bestaan er verschillende padplanners die deze 2 principes elk op hun manier gebruiken/combineren. In dit hoofdstuk worden deze padplanners besproken en vergeleken qua kenmerken. Ook wordt er een opsomming gemaakt van de belangrijkste parameters die nodig zijn voor het padplanningsalgoritme.

2.3.1 OMPL

Open Motion Planning Library of kortweg OMPL implementeert de basis principes van sampling gebaseerde padplanning. OMPL is een bibliotheek van verschillende algoritmes die snel en efficiënt een oplossing kunnen vinden. Bij een omgeving met obstakels wordt er een oplossing gevonden, maar als er geen pad mogelijk is in de omgeving kan OMPL dit niet rapporteren [6], [17].

De bibliotheek OMPL bestaat uit verschillende types algoritmes die hieronder worden opgesomd.

- **Multi-query planners:** sampling gebaseerd algoritme dat één thread gebruikt om een map aan te maken, terwijl een tweede thread controleert of er een pad bestaat in de map tussen begin- en eindtoestand. Een voorbeeld hiervan is PRM.
- **Single-query planners:** deze planners creëren meestal een boomstructuur van geldige toestanden/bewegingen van de robot. Sommige padplanners gebruiken bijvoorbeeld ook twee bomen: één vanaf de begin- en één vanaf de eindtoestand. Zulke planners verbinden dan de start-boom met de eind-boom. Een voorbeeld hiervan is RRT dat in de vorige sectie al werd besproken.
- **Optimizing planners:** in de afgelopen jaren zijn verschillende sampling gebaseerde padplanners ontwikkeld die nog extra optimalisatie garanties bieden voor het gegenereerde pad. Normaal wordt het gevonden pad aangenomen als het kortste en optimale pad. In OMPL kunnen ze kostenfuncties linken aan de omgeving. Zo kan het algoritme hier dan een optimaler pad berekenen i.f.v de omgeving. Twee voorbeelden hiervan zijn PRMStar en RRTStar [18].

OMPL heeft algemene parameters in zijn bibliotheek. Deze kunnen aangepast worden afhankelijk van de omgeving en robot.

- **Longest valid segment fraction:** is de fractie/afstand van de toestandsruimte tussen de knooppunten (van de robot). Als deze een waarde heeft van bijvoorbeeld 0.01, dan zal de afstand tussen de knooppunten kleiner zijn dan 1/100ste t.o.v. de toestandsruimte. Dit is dus de waarde die bepaalt hoe groot de resolutie is voor objecten in de omgeving. Bijvoorbeeld als deze waarde laag is zal de botsingscontrole erg langzaam zijn. Terwijl er bij een hoog ingestelde waarde botsingen rond kleine/smalle objecten gemist worden.
- **Maximum waypoint distance:** is de afstand van het knooppunt tot de rand van het obstakel dat niet wordt gecontroleerd als mogelijke pad. Als deze waarde bijvoorbeeld 0.1 is t.o.v. de toestandsruimte en de afstand tussen een knooppunt en een obstakel is kleiner dan die 0.1, dan berekent het algoritme geen pad via dat knooppunt [6].

2.3.2 CHOMP

Covariant Hamiltonian Optimization for Motion Planning of kortweg CHOMP is een optimalisatie gebaseerde padplanner die snel op de omgeving/obstakels reageert en tegelijkertijd dynamische grootheden zoals snelheid en versnelling optimaliseert [4]. Het convergeert snel naar een soepel, botsingsvrij traject dat efficiënt op de robot kan worden uitgevoerd. Hoewel CHOMP in staat is om zelf een initieel pad te genereren, meestal een rechte lijn van start naar doel [19], zal het geoptimaliseerde pad dat hieruit volgt vaak nog niet optimaal zijn. De oorzaak hiervan ligt bij de lokale minima waarin het algoritme vast komt te zitten. Een alternatief is een sampling gebaseerde planner uit OMPL te gebruiken als initieel pad en dit verder te laten optimaliseren door CHOMP.
Om het pad effectief te berekenen maakt CHOMP gebruik van een kostenfunctie. Er wordt naar een minimum van deze kostenfunctie gezocht door gebruik te maken van gradiënt-vectoren. Deze (negatieve) gradiënt wijst altijd in de richting van de steilste helling naar beneden. Hierdoor wordt de gemiddelde kost snel verlaagd, wat het gegenereerde pad bevoordeeld. Om te controleren op botsingen wordt er gebruik gemaakt van een "signed distance field". Dit is een 3D-map die de afstanden tot de obstakels bevat voor elke positie van de robot in de omgeving. Als deze afstand bijvoorbeeld negatief is, bevindt de robot zich in het object. Dit wordt op voorhand berekend en tijdens het plannen gebruikt om obstakels te vermijden [19].

CHOMP heeft optimalisatieparameters aan zijn algoritme toegevoegd. Deze kunnen aangepast worden voor de gegeven omgeving/robot waarmee gewerkt wordt.

- **Planning time limit:** de maximale tijd die het algoritme kan nemen om een oplossing te vinden voordat deze wordt onderbroken.
- **Max iterations**: het maximale aantal iteraties dat het algoritme kan gebruiken om een goede oplossing te vinden tijdens het optimaliseren van het gegenereerde pad.
- **Max iterations after collision free:** de maximaal aantal iteraties die in het algoritme worden uitgevoerd nadat het pad botsingsvrij is.
- **Obstacle cost weight:** het gewicht dat wordt gegeven aan de obstakels voor het algoritme. Zo zou bijvoorbeeld bij een waarde van 0 de obstakels genegeerd worden en bij waarde 1 zou het een constraint zijn.
- **Learning rate:** de hoeveelheid energie die het algoritme zal gebruiken om toch nog een lokaal minimum te herkennen in de omgeving.
- **Ridge factor:** door de toevoeging van ruis (bepaalde afwijking van het pad) maakt het CHOMP mogelijk om obstakels te vermijden. Dit kan soms ten koste gaan van de vloeiendheid in de beweging van het gegenereerde pad.
- **Collision clearance:** de minimale afstand die moet worden gehandhaafd om obstakels te vermijden.
- **Trajectory initialization method:** hier kan het type initialisatietraject worden opgegeven.
    - **Quintic-spline/linear/cubic:** is de interpolatiemethode die wordt gebruikt voor trajectinitialisatie tussen start- en doeltoestand.
    - **Fill traject:** deze biedt een optie om het traject te initialiseren vanaf een pad dat is berekend op basis van een voorgaande planner, bijvoorbeeld OMPL [4].

De standaardparameters voor CHOMP werken goed in omgevingen zonder obstakels. Bij obstakels gaat CHOMP met standaardparameters meestal vastlopen in lokale minima. Door deze parameters aan te passen, worden de obstakels vermeden en wordt de kwaliteit van het gegenereerde pad verbeterd [4].

2.3.3 STOMP

Sampling gebaseerde padplanners werken goed in complexe omgevingen, toch is er nog vraag naar een ander alternatief omdat het pad vaak een niet-vloeiend verloop heeft. Hierdoor kan er bijvoorbeeld schade ontstaan aan actuatoren omdat de robot niet-natuurlijke acties uitvoert. Stochastic Trajectory Optimization for Motion Planning of kortweg STOMP is een padplanner die vooral bedoeld is om optimalisatie uit te voeren [3]. Er wordt eerst een pad gegenereerd met een sampling gebaseerde planner en vervolgens wordt het verder geoptimaliseerd door STOMP. De aanpak van het algoritme is gebaseerd op het genereren van ruistrajecten op het originele pad, die vervolgens worden geoptimaliseerd en gecombineerd om een bijgewerkt botsingsvrij traject te genereren. Een voordeel van STOMP is dat de optimalisatie rekening houdt met aanvullende functies zoals koppel- en energiebegrenzingen [3].

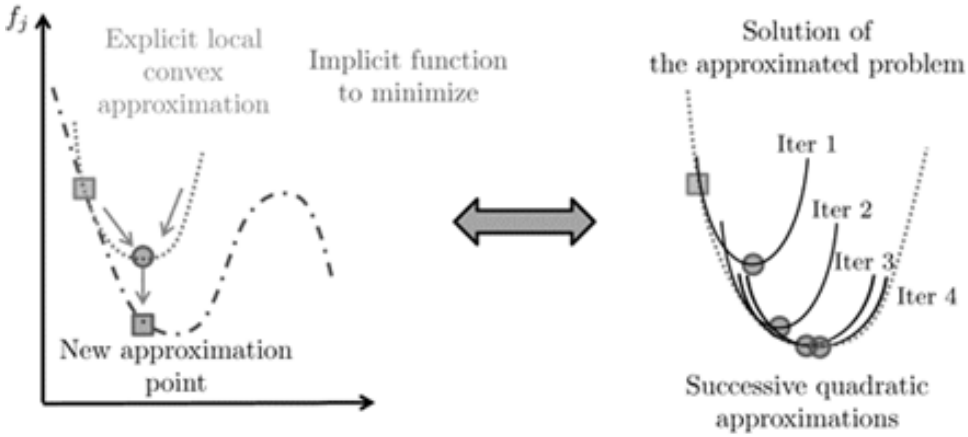
STOMP heeft optimalisatieparameters aan zijn algoritme toegevoegd. Deze kunnen aangepast worden voor de gegeven omgeving/robot waarin gewerkt wordt.

- **Number of timesteps:** de tijd die het algoritme kan nemen om een oplossing te vinden voordat deze wordt onderbroken.
- **Number of iterations:** het aantal iteraties dat het algoritme kan gebruiken om een goede oplossing te vinden tijdens het optimaliseren.
- **Number of iterations after valid:** het maximaal aantal iteraties dat wordt uitgevoerd nadat het pad botsingsvrij is.
- **Noise generator parameters:** de maat van ruis die op de gewrichten wordt toegepast, m.a.w. een grotere waarde komt overeen met een grotere beweging van het ingestelde gewricht [3].

STOMP is een algoritme dat lokale optimalisatie uitstekend uitvoert, d.w.z. dat het beter een lokaal traject kan vinden i.p.v. een globaal traject in de omgeving. De prestaties van STOMP variëren dus afhankelijk van het initiële traject [20]. De standaardparameters voor STOMP werken echter toch goed in de meeste omgevingen. Door de “noise generator parameters” te verhogen kan STOMP beter presenteren in complexe omgevingen met obstakels. STOMP genereert vloeiende, correcte en obstakelvrije bewegingspaden in een relatief korte tijd en kan, volgens experimenten uitgevoerd door de auteurs van STOMP, beter lokale minima vermijden dan CHOMP [3].

2.3.4 TrajOpt

TrajOpt is één van de latere ontwikkelingen (2018) op gebied van optimalisatie gebaseerde padplanners. CHOMP daarentegen werd al uitgebracht in 2009 en STOMP in 2011. Deze planner maakt in tegenstelling tot CHOMP geen gebruik van gradiënt gebaseerde afdaling, maar gebruikt in de plaats sequentiële convexe optimalisatie. Hierbij wordt de kostenfunctie in een bepaald punt benaderd door een parabool waarvan het minimum bepaald wordt. In de buurt van dit minimum wordt er vervolgens een nieuwe benaderingsparabool berekend waarvan weer het minimum bepaald wordt, na een aantal iteraties zal het minimum van de benaderingsparabool overeenkomen met het minimum van de niet convexe functie. In Figuur 11 wordt dit principe weergegeven [21].



Figuur 11: Sequentiële convexe optimalisatie [22, p. 832].

Het controleren op botsingen gebeurt bij TrajOpt door alle obstakels te benaderen met convexe vormen die dan vervolgens gecontroleerd worden op botsingen. Dit gebeurt continu tijdens het plannen waardoor TrajOpt dus kan reageren op een bewegende omgeving. Een ander voordeel is dat TrajOpt ook de mogelijkheid biedt om eenvoudig constraints toe te voegen. Dit zorgt ervoor dat TrajOpt toepasbaar is op een groter bereik aan padplanningsproblemen [21].

Het grote voordeel van deze planner is dat hier op een relatief eenvoudige manier, constraints kunnen gedefinieerd worden voor de joint posities en joint snelheden. De standaardparameters voor TrajOpt werken goed in omgevingen met en zonder obstakels. TrajOpt genereert vloeiende, correcte en obstakelvrije bewegingspaden in een korte tijd en kan anticiperen op bewegende obstakels. Het vermijdt ook lokale minima in tegenstelling tot CHOMP. Het is echter wel nog steeds in ontwikkeling en wordt daarom voortdurend aangepast [21].

2.4 Conclusie

De conclusies van deze literatuurstudie worden samengevat in Tabel 1 en Tabel 2. In deze twee tabellen kunnen de verschillende theoretische kenmerken worden vergeleken. Een "+" betekent dat de planner het beschreven kenmerk goed kan behandelen. Een "-" daarentegen betekent dat de planner niet geschikt is voor het kenmerk.

Tabel 1: Overzicht van het type algoritme dat gebruikt wordt bij de verschillende padplanners.

	OMPL	CHOMP	STOMP	TrajOpt
Sampling gebaseerde padplanning	x			
Optimalisatie gebaseerde padplanning		x	x	x

Tabel 2: Overzicht van kenmerken bij de verschillende padplanners uit de literatuurstudie.

	OMPL	CHOMP	STOMP	TrajOpt
Berekening pad met **standaardparameters** in omgeving **zonder obstakels**	+	+	+	+
Berekening pad met **standaardparameters** in omgeving **met obstakels**	+	-	-	+
Berekening pad met **bijgewerkte parameters** in omgeving **met obstakels**	+	+	+	+
Extra: Kan reageren op bewegende obstakels in omgeving	-	-	-	+
Extra: Feedback algoritme als er geen mogelijke oplossing is	-	+	+	+
Extra: Mogelijkheid om constraints te gebruiken	-	-	-	+
Extra: Beschikbare documentatie voor de planner	+	+	+	-

3 Simulatieomgeving

Het softwarepakket waarin gesimuleerd wordt heet RVIZ. Dit is een softwareprogramma dat gebruikt wordt in combinatie met MoveIt! en ROS Kinetic.

Om een robot te configureren is een package nodig in een catkin-workspace. Deze workspace moet eerst geïnstalleerd worden in combinatie met MoveIt! en ROS Kinetic.
Volgende commando's moeten in de terminal worden ingegeven om ROS en Catkin te installeren.

```
rosdep update
sudo apt-get update
sudo apt-get dist-upgrade

sudo apt-get install ros-kinetic-catkin python-catkin-tools
```

De volgende commando's voor het installeren van "MoveIt!".

```
sudo apt install ros-kinetic-moveit
```

Als slot nog de catkin workspace aanmaken.

```
mkdir -p ~/ws_moveit/src
```

De workspace is aangemaakt en vervolgens kunnen packages van GitHub worden geïnstalleerd.

```
cd ~/ws_moveit/src
git clone -b kinetic-devel [GitHub-link naar package]
```

Na het installeren van packages moet de workspace gebuild worden.

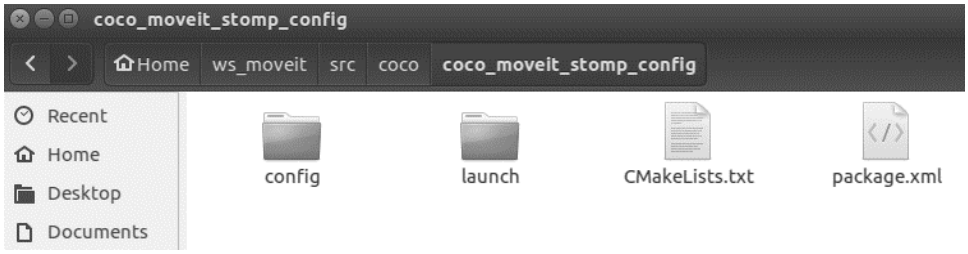
```
cd ~/ws_moveit
catkin config –extend /opt/ros/kinetic
catkin build
```

Als slot moeten deze aanpassingen in de catkin workspace actief gemaakt worden.

```
source ~/ws_moveit/devel/setup.bash
```

Dit laatste commando moet ook toegevoegd worden aan de .bashrc-file, zodat voor elke nieuwe terminal de workspace wordt gesourced.

De workspace is geïnstalleerd, maar de package voor de robot zelf is nog niet aangemaakt. Deze bevindt zich in de src-map. Deze package kan geconfigureerd worden door middel van “MoveIt! Setup Assistant”. Hierin staan dus alle instellingen van de robot om te simuleren. In het volgende hoofdstuk wordt deze tool uitgelegd. Na het instellen van alle parameters in deze tool ziet een package voor een robot er bijvoorbeeld uit zoals weergegeven in Figuur 12.



Figuur 12: Voorbeeld robotpackage in de catkin-workspace

De config-map in Figuur 12 bevat alle instellingen voor de planners die besproken zullen worden in deze masterproef. Deze parameters zijn tevens altijd aanpasbaar. In de launch-map bevinden zich launch-files om de robot met RVIZ op te starten.

De launch-file laadt ook een urdf-bestand in. Hier bevinden zich alle eigenschappen van de robot qua kinematica. In Figuur 13 een voorbeeld hoe één link en de bijhorende joint worden gedeclareerd in het urdf-bestand.

```
<link name="${prefix}base_link" >
  <visual>
    <geometry>
      <mesh filename="package://robot_config/meshes/coco/Base.stl" />
    </geometry>
    <material name="LightGrey">
      <color rgba="0.7 0.7 0.7 1.0"/>
    </material>
  </visual>
  <collision>
    <geometry>
      <mesh filename="package://robot_config/meshes/coco/Base_collision.stl" />
      <scale>0.001 0.001 0.001</scale>
    </geometry>
  </collision>
  <xacro:cylinder_inertial radius="${base_radius}" length="${base_total_length}" mass="${base_mass}">
    <origin xyz="0.0 0.0 ${base_total_length/2}" rpy="0 0 0" />
  </xacro:cylinder_inertial>
</link>

<joint name="${prefix}wheel_turn_joint" type="revolute">
  <parent link="${prefix}base_link" />
  <child link = "${prefix}wheel_link" />
  <origin xyz="0.0 -0.115 0.7445" rpy="0.0 0.0 0.0" />
  <axis xyz="0 1 0" />
  <xacro:unless value="${joint_limited}">
    <limit lower="${wheel_turn_min}" upper="${wheel_turn_max}" effort="150.0" velocity="${0.0007853982*speed_factor}"/>
  </xacro:unless>
  <xacro:if value="${joint_limited}">
    <limit lower="${wheel_turn_min}" upper="${wheel_turn_max}" effort="150.0" velocity="${0.0007853982*speed_factor}"/>
  </xacro:if>
  <dynamics damping="0.0" friction="0.0"/>
</joint>
```

Figuur 13: Voorbeeld van één link en bijhorende joint in een urdf-bestand.

In Figuur 13 wordt eerst een link gedefinieerd als basis. Vervolgens wordt een joint aangemaakt die een “parent” en een “child” heeft. De nieuwe joint heeft als “parent” de basis-link en als “child” een volgende link, die nog aangemaakt moet worden. Zo wordt voor elke joint en link een eigenschap aangemaakt in dit bestand.

In deze masterproef worden 3 robots onderzocht en besproken in volgende deelhoofdstukken.

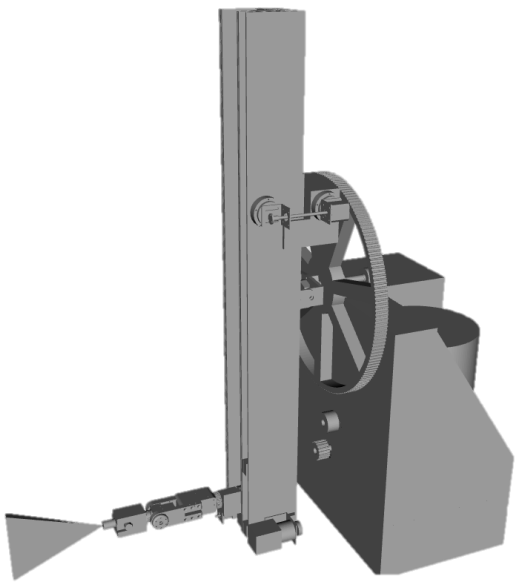
3.1 COCO-verfrobot

De eerste robot die besproken wordt in deze masterproef is een verfrobot. Deze robot werd ontworpen door Gaëtan Auvray. Hij wil deze schilderrobot commercialiseren om schilders hun job te vergemakkelijken. De arm van deze robot heeft een groot bereik als deze helemaal is uitgestrekt, zo moet er geen ladder meer worden gebruikt. De eindeffector van deze robot is een spuitkop.

Het is tevens erg belangrijk dat de eindeffector met een constante snelheid beweegt. Zo wordt de verf gelijkmatig verdeeld over het oppervlak van bijvoorbeeld een muur.

Het doel van deze case is om de omgeving in te lezen, een zigzagbeweging te plannen (voor het verven) en obstakels te vermijden.

De urdf-bestanden heeft Gaëtan Auvray ontworpen. De verfrobot heeft 8 vrijheidsgraden/joints en wordt weergegeven in Figuur 14.



Figuur 14: COCO-verfrobot met spuitkop.

De COCO-robot heeft 3 prismatische gewrichten. Deze kunnen dus enkel lineair bewegen. De andere 5 joints zijn roterende gewrichten. Het eerste gewricht zorgt dat de hoofdarm kan ronddraaien. Het tweede laat de arm toe om naar voor of naar achter te kantelen. In het polsgewricht van de spuitkop bevinden zich de overige drie gewrichten.

3.2 Kuka KR5

De tweede robot die besproken wordt in deze masterproef is de KUKA KR5-robot. De eindeffector van deze robot is een laspistool. Het doel van deze case is om obstakels toe te voegen aan de omgeving en een beweging te plannen van A naar B. Ook hier moeten de planners de toegevoegde obstakels vermijden. De KR5-robot staat ter beschikking op ACRO. Deze robot wordt in het kader van een project gebruikt om met padplanning te kunnen robotlassen.

De urdf-bestanden zijn verkregen via GitHub [23]. De robot heeft 6 vrijheidsgraden en wordt weergegeven in Figuur 15.

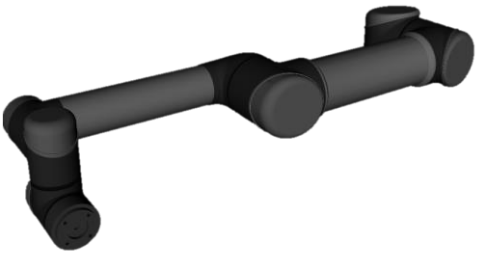
Figuur 15: KUKA KR5-robot met laspistool.

3.3 Universal Robot UR5

Als laatste robot wordt de Universal Robot Type 5 of kortweg UR5-robot besproken. Deze robot heeft geen eindeffector in de simulatie. Het doel van deze case is ook om obstakels toe te voegen aan de omgeving en een beweging te plannen van A naar B. Ook hier moeten de planners de toegevoegde obstakels vermijden. Zo'n UR5-robot is een collaboratieve industriële robotarm. De robot is uitgerust met sensoren die de stromen van elke joint meten. Als de stroom groter is dan een verwachtte waarde, wordt verondersteld at er een botsing is opgetreden. Daarom is deze robot geschikt om naast mensen te werken. Als de robot botst tijdens het uitvoeren van een traject zal dit gedetecteerd worden en stopt de robot.

De UR5-robot staat ter beschikking op ACRO. Voor dit type robot worden de gegenereerde trajecten ook op een echte opstelling getest. Zo kunnen de in simulatie gegenereerde trajecten gecontroleerd worden of ze wel degelijk uitvoerbaar zijn.

De urdf-bestanden zijn verkregen via GitHub [24]. De robot heeft 6 vrijheidsgraden en wordt weergegeven in Figuur 16.



Figuur 16: Universal Robot UR5.

3.4 MoveIt! Setup Assistant

Om een robot te simuleren is een package nodig waar alle instellingen in verwerkt zijn. Deze map/package wordt geconfigureerd met “MoveIt! Setup Assistant”.
De kenmerken/eigenschappen van een robot staan origineel in een urdf-bestand. Dit bestand wordt teruggevonden op GitHub, voor zover beschikbaar.

De Setup Assistant wordt opgestart met het volgende commando in de terminal.

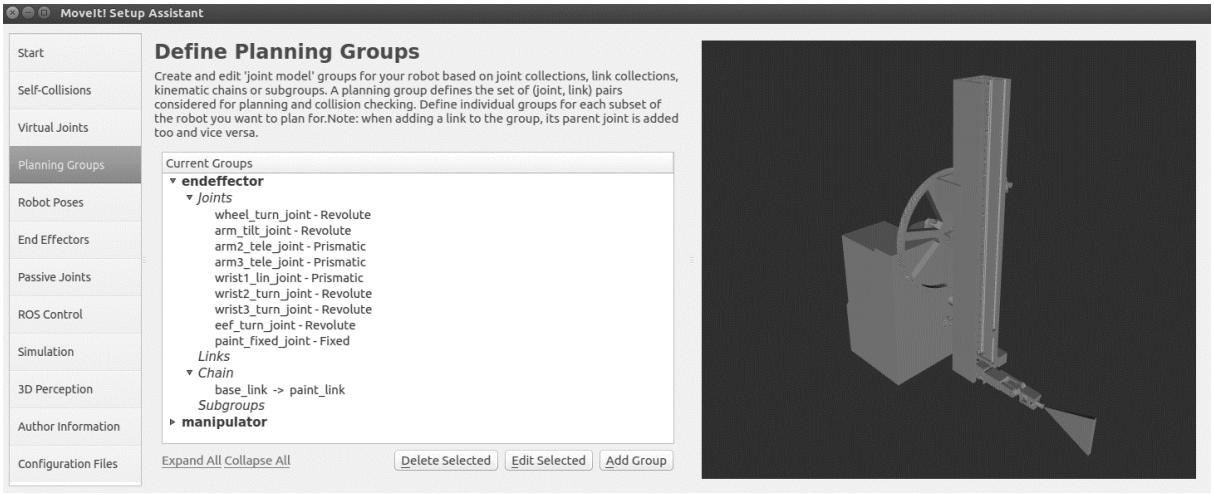
```
roslaunch moveit_setup_assistant setup_assistant.launch
```

Vervolgens wordt volgende wizard gestart, zie Figuur 17.

Figuur 17: Startscherm van Moveit! Setup Assistant.

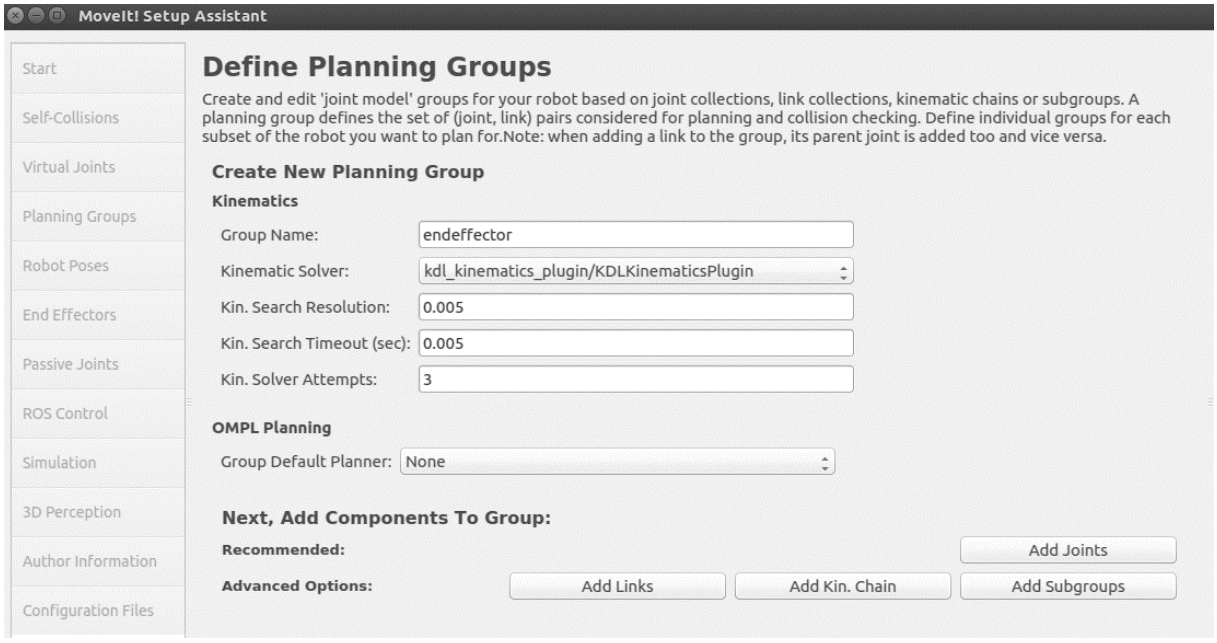
In Figuur 17 kan gekozen worden om een bestaande package te veranderen of een nieuwe package aan te maken met het opgehaalde urdf-bestand van GitHub.

De belangrijkste eigenschap van deze wizard is het instellen van de planning-group. Deze bevat de relatie tussen joints en links nodig voor planning en obstakeldetectie van de robot. In Figuur 18 wordt de planning-group “endeffector” toegevoegd aan de nieuwe package.



Figuur 18: Instellen van een planning groep in MoveIt! Setup Assistant.

In Figuur 19 wordt ook de kinematische oplosser gedefinieerd voor de planning groep. De default oplosser voor robots is de kdl-plugin.



Figuur 19: Definiëren van een kinematische oplosser in MoveIt! Setup Assistant.

In Figuur 19 wordt ook nog een kinematische ketting van links toegevoegd. Zo kan het begin en einde van de robotlinks gedefinieerd worden. Op het einde van deze wizard moet de package worden opgeslagen. Dit gebeurt altijd in de src map binnenin de workspace. De naam van de package is belangrijk, want deze is niet meer aanpasbaar achteraf.

4 Opbouwen van scènes in RVIZ

Om de planners te testen is er een omgeving voor de robot nodig, deze kan op 2 verschillende manieren bekomen worden. De eerste mogelijkheid is door zelf een omgeving aan te maken met eenvoudige kubussen, cilinders en dergelijke. Deze worden “collision objects” of obstakels genoemd. Een tweede manier is het inlezen van de werkelijke omgeving door middel van sensoren. Het resultaat van deze sensoren is een puntenwolk welke als obstakelobject in de simulatieomgeving kan worden toegevoegd. Beide principes voor het invoegen van een omgeving worden in dit hoofdstuk besproken.

4.1 Manueel invoegen van obstakels in de robotomgeving

Voor een academische evaluatie van een planner in simulatie volstaat het om zelf een eenvoudige omgeving aan te maken. Dit is mogelijk door obstakels toe te voegen aan de “planning scene”. Dit toevoegen gebeurt met een Python programma dat geïntegreerd wordt in de package “kr5_moveit_config”. Deze visualiseert de robot en start de planner. Een deel van het programma “collision_scene.py”, dat de objecten toevoegt, is weergegeven in Figuur 20. Hier worden 2 obstakels toegevoegd, de grond en de paal. De toegevoegde obstakels worden weergegeven in Figuur 21.

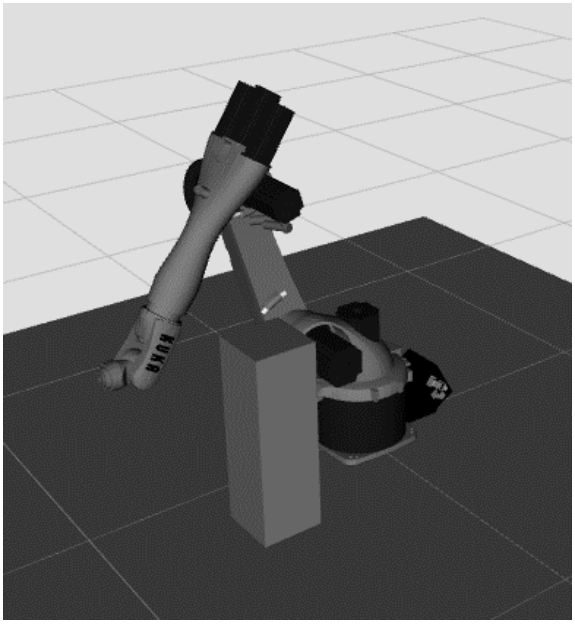
```
def add_world(self):
# ground
    box1_pose = [0, 0, -0.5, 0, 0, 0, 1]
    box1_dimensions = [3, 3, 1]

    box2_pose = [0.5, 0.25, 0, 0, 0, 0, 1]
    box2_dimensions = [0.2, 0.2, 1.25]

    self.add_box_object("ground", box1_dimensions, box1_pose)
    self.add_box_object("object_1", box2_dimensions, box2_pose)

    print "========== Added world to the scene!!"
```

Figuur 20: Code van het programma collision_scene om 2 obstakels toe te voegen.



Figuur 21: KR5-robot in zijn obstakelomgeving.

4.2 Implementatie van Kinect v2

Het manueel opbouwen van een robotomgeving is minder interessant om de planners te evalueren. Het is interessanter om de planner te evalueren in een omgeving gescand door sensoren.

De camera die bij deze masterproef gebruikt werd om de omgeving te scannen is een Kinect v2. Ondanks dat deze sensor ontwikkeld is voor een spelconsole kent ze toch veel gebruik in het roboticaonderzoek. De Kinect v2 is gemakkelijk in gebruik en vooral goedkoop in vergelijking met andere sensoren. Hij heeft een kleuren-, diepte- en infraroodsensor. Doordat deze sensor vaak gebruikt werd in andere onderzoeken, is er veel ondersteunende software ontwikkeld en beschikbaar.

De eerste stap om gebruik te maken van een Kinect v2 camera is het installeren van de juiste drivers. Deze software maakt het mogelijk om de data van de camera in te lezen op een computer. De meest voorkomende drivers zijn OpenKinect en Libfreenect2. In deze masterproef werd gekozen om gebruik te maken van de driver Libfreenect2. Hiervoor is een tutorial beschikbaar, die het gebruik en de implementatie van Libfreenect2 vereenvoudigt [25].

Om de driver te installeren moet de GitHub repository eerst en vooral naar de "home folder" van het Linux computersysteem gekopieerd worden. Dit gebeurt via onderstaand commando in de terminal.

```
git clone https://GitHub.com/OpenKinect/libfreenect2.git
```

Om de nodige drivers te kunnen installeren is het nodig om de build tool "cmake" eerst te installeren op de computer. Dit gebeurt met onderstaand commando.

```
sudo apt-get install build-essential cmake pkg-config
```

Vervolgens zijn er nog de drivers voor de USB-poorten en grafische kaart nodig. Deze worden met onderstaande commando's geïnstalleerd.

```
sudo apt-get install libusb-1.0-0-dev
sudo apt-get install libturbojpeg libjpeg-turbo8-dev
sudo apt-get install libglfw3-dev
```

Na deze installatie kan de Libfreenect2 volledig geïnstalleerd worden. Hiervoor moet er een map "build" in de package Libfreenect2 aangemaakt worden. Daarna kan de package geïnstalleerd worden. Dit wordt uitgevoerd via onderstaande commando's.

```
mkdir build && cd build
cmake .. -DCMAKE_INSTALL_PREFIX=$HOME/freenect2
make
make install
```

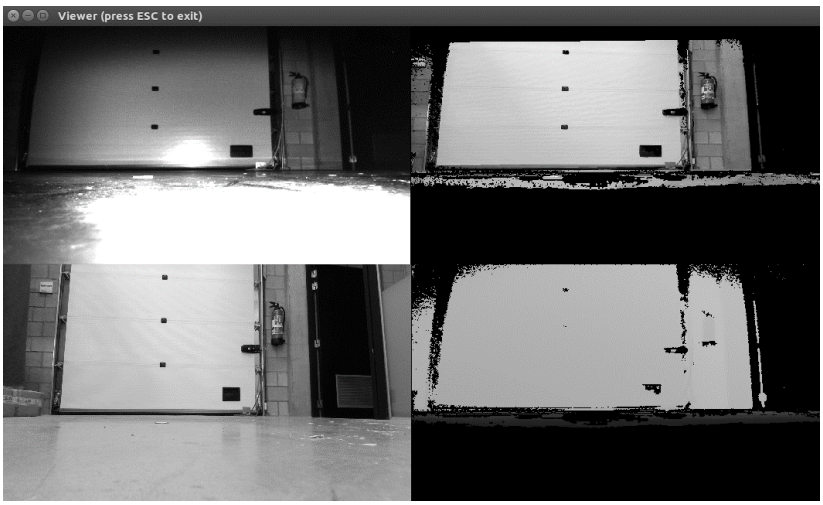
De laatste stap, om de Kinect v2 te gebruiken, is het toevoegen van code zodat de camera gebruikt kan worden zonder een administrator toegang. Hiervoor wordt het onderstaand commando gebruikt.

```
sudo cp ../platform/linux/udev/90-kinect2.rules /etc/udev/rules.d/
```

Nu alles geïnstalleerd is kan de Kinect v2 getest worden met onderstaand commando. Dit commando wordt opgeroepen vanuit de "build" map van Libfreenect2.

```
./bin/Protonect
```

In Figuur 22 worden livebeelden getoond van de Kinect v2 via dit testprogramma.



Figuur 22: Voorbeeld van het testprogramma Protonect.

De data van de Kinect v2 is nu beschikbaar op de computer maar kan nog niet gebruikt worden in ROS. Hiervoor is er nog een package nodig dat de data van Libfreenect2 doorgeeft aan ROS. Deze package is iai_kinect2 [26].

Eerst moet de GitHub repository van iai_kinect2 gekopieerd worden naar de “src” map van de catkin workspace waarbinnen de Kinect v2 gebruikt zal worden. Vervolgens kan de package geïnstalleerd worden, dit gebeurt via onderstaande commando’s.

```
cd $YOUR_CATKIN_WORKSPACE_PATH/src
git clone https://GitHub.com/code-iai/iai_kinect2.git
cd iai_kinect2
rosdep install -r --from-paths .
cd ~/YOUR_CATKIN_WORKSPACE_PATH
catkin_make -DCMAKE_BUILD_TYPE="Release"
```

Na deze installatie kan het programma in deze package getest worden met onderstaand commando.

```
roslaunch kinect2_bridge kinect2_bridge.launch
```

Na het uitvoeren van dit commando zal er in de terminal de regel “Waiting for clients to connect” weergegeven worden. Dit wil zeggen dat alles correct werkt en het programma nu aan het wachten is tot er een package binnen ROS de data van de Kinect v2 begint te gebruiken.

Op bepaalde computers komen er, met bovenstaande configuratie, nog foutmeldingen voor. Meestal ligt de oorzaak hiervan bij de grafische kaart van de computer. Deze kan mogelijk niet compatibel zijn met de gebruikte programma’s of de drivers hiervan zijn niet bijgewerkt. Een eenvoudige manier om deze foutmeldingen te omzeilen is het gebruik van onderstaand commando. Hierdoor zal de grafische kaart niet meer instaan voor de verwerking van de beelden, maar wel de CPU. Doordat de beeldverwerking over de CPU gaat, vertraagt de computer aanzienlijk.

```
roslaunch kinect2_bridge kinect2_bridge.launch reg_method:=cpu depth_method:=cpu
```

Het programma publiceert de data van de Kinect v2 nu op een aantal verschillende topics binnen ROS, met elk hun eigenschappen. Zo zijn er topics voor HD beelden, SD beelden, kleuren beelden of enkel diepte-informatie van die beelden. Alle actieve topics kunnen weergegeven worden met het volgende commando.

```
rostopic list
```

Nu alle data beschikbaar zijn op de computer, kan verder gegaan worden met het omvormen van deze data naar obstakels voor de robotomgeving.

4.3 Puntenwolk omvormen naar mesh

De meeste sensoren die gebruikt worden om een omgeving in te lezen, stellen deze digitaal voor als een puntenwolk. Hierbij heeft elk punt een xyz-coördinaat en eventueel zelfs een kleur. Een puntenwolk bevat vaak miljoenen punten om bijvoorbeeld een vlak voor te stellen. Dit zijn veel data die verwerkt moet worden en zou voor vertraging kunnen zorgen. Om de hoeveelheid data te beperken is er gekozen om de puntenwolk om te vormen naar een mesh. Het maken van deze mesh gebeurt door elk punt met elkaar te verbinden om zo één 3-dimensionaal object te verkrijgen. In dit deelhoofdstuk wordt er verklaard hoe zo'n mesh bekomen kan worden.

Binnen deze masterproef is er gekozen om met het programma CloudCompare te werken. Deze software is specifiek ontwikkeld om allerlei bewerkingen op puntenwolken uit te voeren. Enkele voorbeelden zijn: het aanpassen van kleureninformatie, vlak fitten, afstanden berekenen, uitschieters uit de data verwijderen, etc. In deze toepassing wordt gebruik gemaakt van de functie "vlak fitten", om een mesh te creëren.

Om de puntenwolk om te vormen naar een mesh moet er eerst één opgeslagen worden. Dit gebeurt met behulp van een rosbag-bestand. Zo'n bestand is een opname van alle data ontvangen van de Kinect v2. Binnen ROS kan dit bestand opnieuw worden afgespeeld. Om een rosbag-bestand te creëren worden volgende commando's gebruikt.

Eerst wordt de Kinect v2 aangesloten op de computer en dient iai_kinect2 gestart te worden om de data door te geven aan ROS. Dit gebeurt via onderstaand commando.

```
roslaunch kinect2_bridge kinect2_bridge.launch
```

Na het opstarten van de iai_kinect2 package worden de data gepubliceerd op het topic "/kinect2/hd/points". Om de data op te slaan in een rosbag-bestand wordt onderstaand commando gebruikt.

```
rosbag record /kinect2/hd/points
```

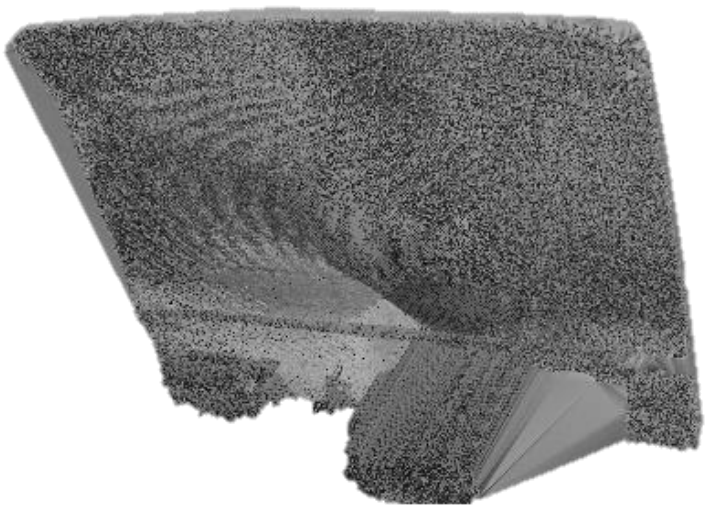
Na enkele seconden kan het opslaan stopgezet worden met de toetsencombinatie "ctrl+c". Het rosbag-bestand is nu nog niet bruikbaar binnen CloudCompare. Hiervoor moet deze eerst omgevormd worden naar een pcd-bestand. Hiervoor is een tool voorzien in de package "pcl_ros". Deze kan opgeroepen worden met het volgende commando.

```
rosrun pcl_ros bag_to_pcd pointcloud.bag /kinect2/hd/points /home/pointclouds /base_link
```

Hierbij is:

- Pointcloud.bag het rosbag-bestand dat eerder opgeslagen werd;
- /kinect2/hd/points is de selectie van het gewenste topic in het rosbag-bestand;
- /home/pointclouds het pad waar het pcd-bestand opgeslagen wordt;
- /base_link het assenstelsel waarin het pcd-bestand opgeslagen wordt.

Het pcd-bestand dat nu aangemaakt is kan worden geopend met CloudCompare om de puntenwolk om te vormen naar een mesh. In het menu “Edit” bevindt zich de optie “mesh” die op zijn beurt de optie “Delaunay 2.5D (best fitting plane)” bevat. Door hierop te klikken wordt een mesh aangemaakt die de puntenwolk omvat. Het resultaat is weergegeven in Figuur 23.



Figuur 23: Mesh aangemaakt door CloudCompare.

De mesh die nu aangemaakt is wordt geëxporteerd als STL-bestand en kan vervolgens gebruikt worden als obstakel in de robotomgeving.

4.4 Mesh toevoegen aan scène

Om een mesh toe te voegen aan de scène in RVIZ wordt gebruik gemaakt van een nieuwe package en een cpp-bestand. In het vorige deelhoofdstuk werd besproken hoe een puntenwolk te transformeren is naar een STL-bestand. Dit bestand kan ingeladen worden als obstakel object in RVIZ. De nieuwe sub-package "add_mesh" werd toegevoegd aan de package "moveit_tutorials". Hier wordt de cpp-code uitgevoerd om het obstakel toe te voegen. In Figuur 24 wordt de code beschreven die uitgevoerd wordt.

```
moveit_msgs::CollisionObject muur;
//mesh halen uit volgende bron:
shapes::Mesh* m = shapes::createMeshFromResource("package://robot_config/meshes/muur.stl");
ROS_INFO("Muur geimporteerd!");
shape_msgs::Mesh mesh_m;
shapes::ShapeMsg wall_mesh_msg;
shapes::constructMsgFromShape(m,wall_mesh_msg);
mesh_m = boost::get<shape_msgs::Mesh>(wall_mesh_msg);
muur.meshes.resize(1);
muur.meshes[0] = mesh_m;
muur.mesh_poses.resize(1);

muur.mesh_poses[0].position.x = -0.5;
muur.mesh_poses[0].position.y = 1.0;
muur.mesh_poses[0].position.z = -1.0;
muur.mesh_poses[0].orientation.w= 1.0;
muur.mesh_poses[0].orientation.x= 0.0;
muur.mesh_poses[0].orientation.y= 0.0;
muur.mesh_poses[0].orientation.z= 0.0;

muur.meshes.push_back(mesh_m);
muur.mesh_poses.push_back(muur.mesh_poses[0]);
muur.operation = muur.ADD;

std::vector<moveit_msgs::CollisionObject> collision_objects;
collision_objects.push_back(muur);

// Nu collision object toevoegen aan scene in RVIZ:
ROS_INFO("Muur toegevoegd aan omgeving!");
planning_scene_interface.addCollisionObjects(collision_objects);
```

Figuur 24: Code van de package add_mesh om STL-bestand als obstakel toe te voegen.

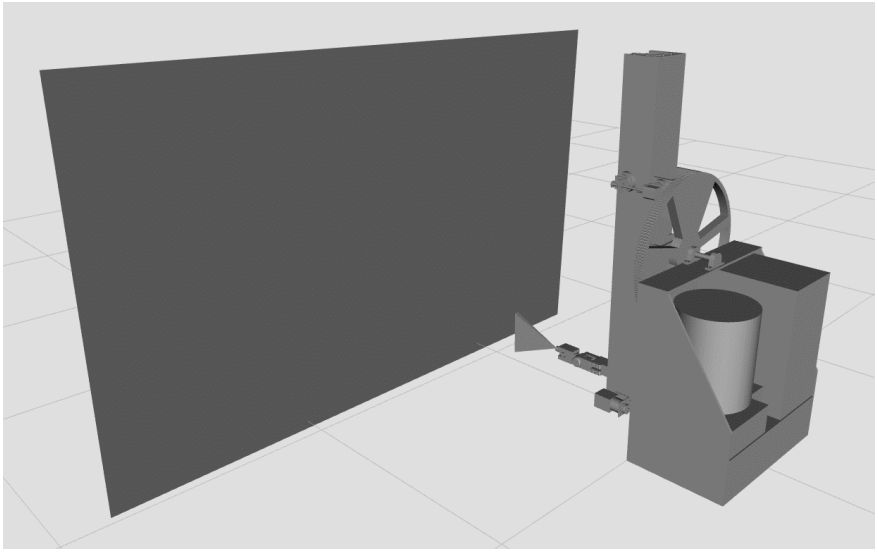
In Figuur 24 wordt code uitgevoerd die eerst een STL-bestand inlaadt, om het vervolgens toe te voegen als obstakel. Om het obstakel toe te voegen moet eerst een "roslaunch" worden uitgevoerd. In dit geval is het de verfrobot die wordt opgestart, omdat het obstakel een muur is.

```
roslaunch coco_moveit_config demo.launch
```

In een nieuwe terminal kan dan de "rosrun" worden uitgevoerd om de package "add_mesh" uit te voeren.

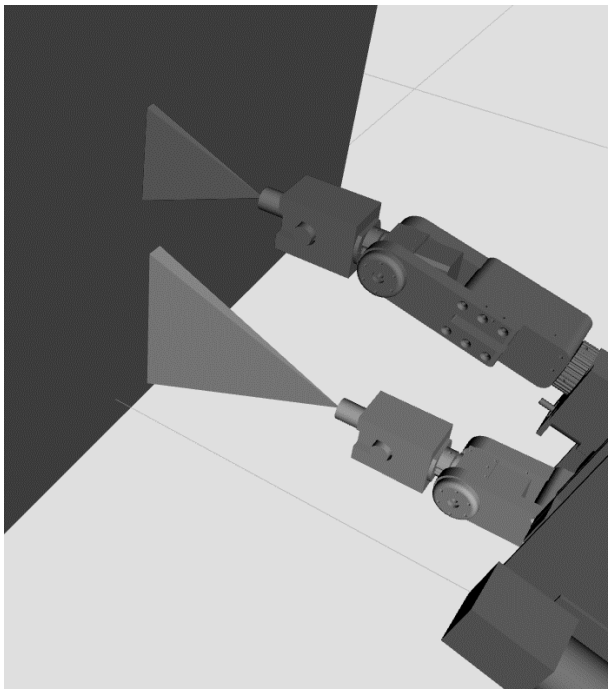
```
rosrun moveit_tutorials add_mesh
```

De muur is nu toegevoegd als obstakel aan de scène in RVIZ. Dit wordt weergegeven in Figuur 25.



Figuur 25: Muur als STL-bestand toegevoegd aan RVIZ.

De planners houden hier nu rekening mee tijdens een beweging van de robotarm door middel van botsingsdetectie. De minimum spuitafstand van de spuitkop wordt ook in rekening gebracht bij de botsingsdetectie. Dit wordt weergegeven in Figuur 26.

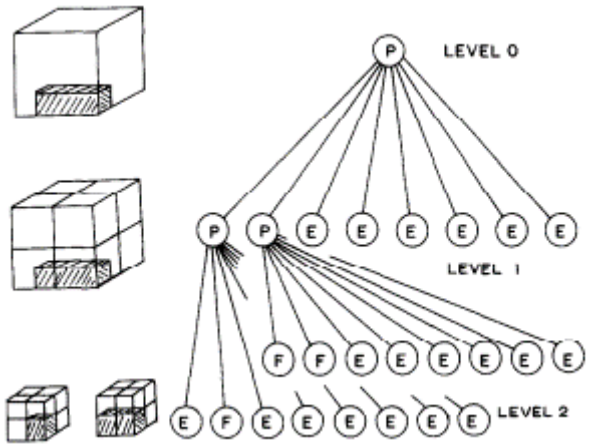


Figuur 26: Botsingsdetectie op de toegevoegde muur in RVIZ.

In Figuur 26 is de onderste spuitkop van de robot in de begintoestand. Deze is niet in botsing met de muur (obstakel). De bovenste spuitkop is de eindtoestand waarnaar de planner zou moeten gaan. Deze is echter wel in botsing met de muur, indien de minimum spuitafstand ook in rekening wordt genomen. Er kan bijgevolg geen pad berekend worden.

4.5 Integratie van octomap

In het vorige hoofdstuk werd een puntenwolk omgevormd naar een mesh. Er bestaat nog een andere manier om met de puntenwolk te werken, namelijk een octree. Zo'n octree is een boomstructuur aan nodes die wordt voorgesteld in Figuur 27.



Figuur 27: Grafische structuur/opbouw van een octree [27, p. 135].

In Figuur 27 wordt eerst een grote kubus/node toegevoegd. Deze (grootste) node bevat alle punten die gescand werden in de puntenwolk. Vervolgens wordt de kubus opgesplitst in acht kleinere kubussen die childnodes genoemd worden. Als zo'n childnode een punt/object bevat wordt deze weer verder opgesplitst in acht kleinere kubussen. Dit proces blijft zich herhalen tot een bepaalde grootte van de kubussen die opgelegd wordt in de instellingen. Door deze methode wordt de gescande omgeving voorgesteld door allemaal kleine kubusjes en is de hoeveelheid data sterk verminderd, aangezien een puntenwolk vaak miljoenen punten voorstelt [27].

Om de puntenwolk van de Kinect v2 om te vormen naar een octree wordt er gebruik gemaakt van de package Octomap [28].

De installatie van Octomap gebeurt met onderstaand commando.

```
git clone https://GitHub.com/OctoMap/octomap.git

cd octomap
mkdir build
cd build
cmake ..
make
```

Octomap is nu geïnstalleerd maar kan nog niet binnen ROS gebruikt worden. Hiervoor dienen de packages “octomap_ros”, “octomap_msgs” en “octomap_mapping” geïnstalleerd te worden. De installatie gebeurt door, voor elke package, de GitHub repository’s te kopiëren naar de src-map van de catkin-workspace. Hiervoor worden onderstaande commando’s gebruikt.

```
cd ws_moveit/src
git clone https://GitHub.com/OctoMap/octomap_ros.git
git clone https://GitHub.com/OctoMap/octomap_mapping.git
git clone https://GitHub.com/OctoMap/octomap_msgs.git
```

De laatste stap is het builden van de workspace. Dit gebeurt met volgende commando’s.

```
catkin build
cd ~/ws_moveit
source devel/setup.bash
```

Om octomap te gebruiken is de subpackage “perception_pipeline” voorzien in de package “moveit_tutorials”. Deze subpackage moet echter wel nog aangepast worden om de data te kunnen gebruiken. In het bestand “obstacle_avoidance.launch” binnen de map “launch” van de package “perception_pipeline” worden volgende aanpassingen gedaan, weergegeven in Figuur 28.

```
3  <launch>
4    <include file="$(find coco_moveit_chomp_config)/launch/demo_chomp.launch" />
5
6    <!-- Play the rosbag that contains the pointcloud data -->
7    <node pkg="moveit_tutorials" type="bag_publisher_maintain_time" name="point_clouds" />
8
9    <!-- If needed, broadcast static tf for robot root -->
10   <node pkg="tf2_ros" type="static_transform_publisher" name="to_temp_link" args="-1.2 1.4 -0.1 0 0 0  temp_link base_link" />
11   <node pkg="tf2_ros" type="static_transform_publisher" name="to_coco_base" args="0 0 0 0 0 1.48 kinect2_rgb_optical_frame temp_link" />
12
13 </launch>
```

Figuur 28: Obstacle_avoidance.launch na de aanpassingen.

In regel 4 van Figuur 28 wordt er aangegeven welke simulatie er opgestart dient te worden. In dit geval is dit de verfrobot met de CHOMP planner. Regel 10 en 11 zorgen ervoor dat de octree op de juiste plaats in de simulatieomgeving komt te staan. Regel 10 zorgt voor een verplaatsing over de x, y en z-as. Regel 11 staat in voor de rotatie rond de x, y en z-as. Belangrijk bij deze transformaties zijn de namen van de assenstelsels waartussen getransformeerd wordt. De basis van de verfrobot is “base_link” en het assenstelsel van de puntenwolk komende van de Kinect v2 is “kinect2_rgb_optical_frame”.

Als tweede moeten er aanpassingen gebeuren in het bestand “bag_publisher_maintain_time.cpp” binnen de map src-map van de package “perception_pipeline”. Het deel van de code dat aangepast dient te worden is weergegeven in Figuur 29. Deze code leest de gegevens van een rosbag die een puntenwolk bevat in, en geeft deze door aan octomap. Het aanmaken van een rosbag werd eerder besproken in 4.3.

```
43  int main(int argc, char** argv)
44  {
45    ros::init(argc, argv, "bag_publisher_maintain_time");
46    ros::NodeHandle nh;
47
48    ros::Publisher point_cloud_publisher = nh.advertise<sensor_msgs::PointCloud2>("/kinect2/hd/points", 1);
49    ros::Rate loop_rate(0.1);
50
51    // Variable holding the rosbag containing point cloud data.
52    rosbag::Bag bagfile;
53    std::string path = ros::package::getPath("moveit_tutorials");
54    path += "/doc/perception_pipeline/bags/poort.bag";
55    bagfile.open(path, rosbag::bagmode::Read);
56
57    std::vector<std::string> topics;
58    topics.push_back("/kinect2/hd/points");
59
```

Figuur 29: Bag_publisher_maintain_time.cpp na aanpassing.

In regel 48 en 58 van Figuur 29 dient het topic waarop de rosbag gepubliceerd wordt ingevuld te worden. Regel 54 bevat het pad waar de rosbag, die uitgelezen dient te worden, opgeslagen is.

De volgende aanpassingen gebeuren in de package van de robot die gebruikt wordt. In dit geval is dit “coco_moveit_config”. Het eerste bestand dat hier aangepast dient te worden is “sensor_manager.launch.xml” welk zich in de map “launch” bevindt. De code in dit bestand is weergegeven in Figuur 30.

```
3   <launch>
4
5     <!-- This file makes it easy to include the settings for sensor managers -->
6
7     <!-- Param for kinect config -->
8     <rosparam command="load" file="$(find coco_moveit_chomp_config)/config/sensors_kinect_pointcloud.yaml" />
9
10    <!-- Params for the octomap monitor -->
11    <!--  <param name="octomap_frame" type="string" value="some frame in which the robot moves" /> -->
12    <param name="octomap_frame" type="string" value="kinect2_rgb_optical_frame" />
13    <param name="octomap_resolution" type="double" value="0.05" />
14    <param name="max_range" type="double" value="5.0" />
15    <param name="use_sim_time" value="true" />
16
17    <!-- Load the robot specific sensor manager; this sets the moveit_sensor_manager ROS parameter -->
18    <arg name="moveit_sensor_manager" default="coco" />
19    <include file="$(find coco_moveit_chomp_config)/launch/$(arg moveit_sensor_manager)_moveit_sensor_manager.launch.xml" />
20
21  </launch>
```

Figuur 30: Sensor_manager.launch.xml na aanpassing.

In Figuur 30 zijn vooral regels 12, 13 en 14 belangrijk. Als eerste definieert regel 12 het assenstelsel waarin de uiteindelijke octree opgebouwd wordt. In regel 13 kan er aangegeven worden wat de gewenste resolutie is voor de octree, deze bepaalt de kleinst mogelijke kubus in de gehele octree. Regel 14 bepaalt de maximale afstand waarbij sensordata gebruikt wordt. Alles voorbij deze ingestelde afstand wordt niet opgenomen in de octree.

De laatste stap in het configureren van octomap is het toevoegen van een bestand genaamd “sensors_kinect_pointcloud.yaml” in de map “config” van de robot. Dit bestand bevat de code weergegeven in Figuur 31.

```
3  sensors:
4    - sensor_plugin: occupancy_map_monitor/PointCloudOctomapUpdater
5      point_cloud_topic: /kinect2/hd/points
6      max_range: 5.0
7      point_subsample: 1
8      padding_offset: 0.1
9      padding_scale: 1.0
10     max_update_rate: 1.0
11     filtered_cloud_topic: filtered_cloud
12 /home/christoph/ws_moveit/src/coco/coco_moveit_chomp_config/config/sensors_kinect_pointcloud.yaml
```

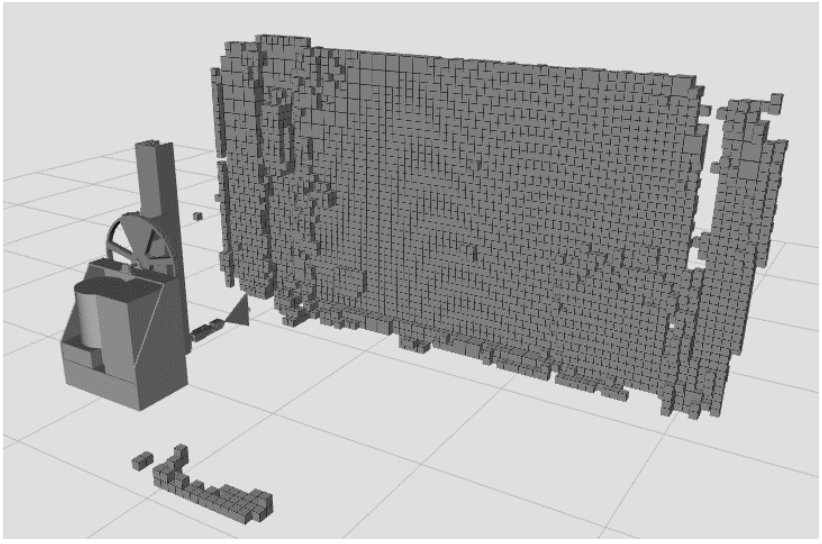
Figuur 31: Code in het bestand "sensors_kinect_pointcloud.yaml".

De belangrijkste regel in Figuur 31 is regel 5, waarin wordt aangegeven op welk topic de puntenwolk, die we later omvormen naar een octree, gepubliceerd wordt. De andere regels in Figuur 31 bevatten instellingen voor het optimaliseren van een octomap, maar deze werden niet aangepast.

Na het correct instellen van alle bestanden kan de originele puntenwolk omgevormd worden tot een octree. Het bestand “obstacle_avoidance.launch” kan worden opgeroepen met onderstaand commando in de terminal.

```
roslaunch moveit_tutorials obstacle_avoidance.launch
```

De simulatie omgeving wordt opgestart met daarin de robot en een octree benaderde voorstelling van de puntenwolk ingelezen door een Kinect v2. Dit is weergegeven in Figuur 32.



Figuur 32: Simulatieomgeving met de gewenste robot en octree.

4.6 Conclusie

Het simulatiepakket “MoveIt!” biedt verschillende mogelijkheden om obstakels toe te voegen aan de robotomgeving, deze werden in dit hoofdstuk besproken en toegepast. Er kan geconcludeerd worden dat elk van de geteste methodes relatief eenvoudig in gebruik is en elk hun eigen voordelen hebben.

De eerste methode, besproken in dit hoofdstuk, is zelf een omgeving aanmaken met obstakels. Zo’n geprogrammeerde omgeving kan eenvoudig of juist complex worden aangemaakt naargelang de toepassing. Dit soort omgeving is ideaal voor academische testen. Deze methode werd gebruikt op de KR5-robot en de UR5-robot, respectievelijk besproken in hoofdstuk 6 en 7.

De tweede methode die besproken werd is het effectief invoegen van een omgeving door gebruik te maken van een sensor. Met behulp van een Kinect v2 kan de omgeving worden ingelezen als een puntenwolk. Eerst werd besproken hoe zo’n puntenwolk wordt omgevormd tot een mesh. Hierdoor gaat slechts een minimale hoeveelheid data van de puntenwolk verloren. Er is echter wel manueel werk vereist om zo’n mesh aan te maken. Vervolgens werd ook een oplossing gerealiseerd door te werken met een octree via Octomap. Het belangrijkste voordeel hier is dat er enkel een aantal instellingen moeten gebeuren en het aanmaken van de octree hierna volledig automatisch gebeurt. Deze methode wordt gebruikt in hoofdstuk 5.

Octomap kan tevens ook nog gebruikt worden om live-beelden van de omgeving in te voegen.

5 Evaluatie van padplanners met de verfrobot “COCO”

Het specifieke aan deze toepassing/case is dat er een contourbeweging gerealiseerd moet worden. In deze case wordt een evaluatie gedaan in welke mate die contourbeweging/zigzagbeweging mogelijk zijn met de genoemde padplanners. Om de zigzagbeweging te realiseren vergelijkt deze masterproef verschillende planners. Tijdens het plannen is het erg belangrijk dat de robot nooit in botsing komt met zijn omgeving en dat de lineaire posities en constante snelheden van de zigzagbeweging gerespecteerd worden.

De planners hebben een aantal parameters nodig tijdens het simuleren van de zigzagbeweging. Eerst wordt de breedte en hoogte ingesteld van het te verven gebied. Ook wordt de openingshoek van de spuitkop ingesteld. Deze is ingesteld op 43,6 graden, wat op een afstand van 0,25 meter tot de muur een spuithoogte van 0,2 meter voorstelt op de muur. De overlap tussen twee opeenvolgende verfbanen tijdens de zigzagbeweging is ingesteld op 50%. De hoek is tevens altijd aanpasbaar om meer verfoppervlakte te creëren in simulatie.

Per deelhoofdstuk worden verschillende planners besproken voor deze robot. Elke planner wordt naar dezelfde start- en eindconfiguraties gestuurd en wordt in identieke omstandigheden/omgeving getest.

5.1 Zigzagbeweging met Cartesische interpolatie

Deze methode werd niet in de literatuurstudie (hoofdstuk 2) besproken, maar is wel een onderdeel in “MoveIt!”. Met deze planner kunnen waypoints of tussenpunten voor een pad worden gedefinieerd. Met behulp van een Python programma wordt de eindeffector van de verfrobot in xyz-coördinaten naar zijn einddoel gestuurd. De Cartesische interpolatie zorgt ervoor dat tussen het opgegeven begin- en eindconfiguratie lineaire interpolatie wordt toegepast om tussenpunten te creëren. Deze tussenpunten zijn ingesteld op een lineaire afstand onderling van 1 millimeter, zie Figuur 33.

```
waypoints = []

# huidige positie inlezen
wpose = move_group.get_current_pose().pose

# bewegen in de gewenste richting (x: -50 cm)
wpose.position.x -= 0.5
waypoints.append(copy.deepcopy(wpose))

# vervolgens bewegen in andere richting (z: +20 cm)
wpose.position.z += 0.2
waypoints.append(copy.deepcopy(wpose))

# We want the Cartesian path to be interpolated at a resolution of 1 mm
# which is why we will specify 0.001 as the eef_step in Cartesian
# translation.  We will disable the jump threshold by setting it to 0.0,
# ignoring the check for infeasible jumps in joint space.
(plan, fraction) = move_group.compute_cartesian_path(
                                   waypoints,   # waypoints to follow
                                   0.001,        # eef_step
                                   0.0)         # jump_threshold

move_group.execute(plan, wait=True)
```

Figuur 33: Definiëren van waypoints voor Cartesische interpolatie.

In Figuur 33 worden geen oriëntaties gedefinieerd. De planner houd deze constant voor alle bewegingen na het inlezen van de huidige positie. De eerste oriëntatie is voor de spuitkop gedefinieerd in de urdf-bestanden. Snelheden en versnellingen worden niet doorgegeven aan de planner.

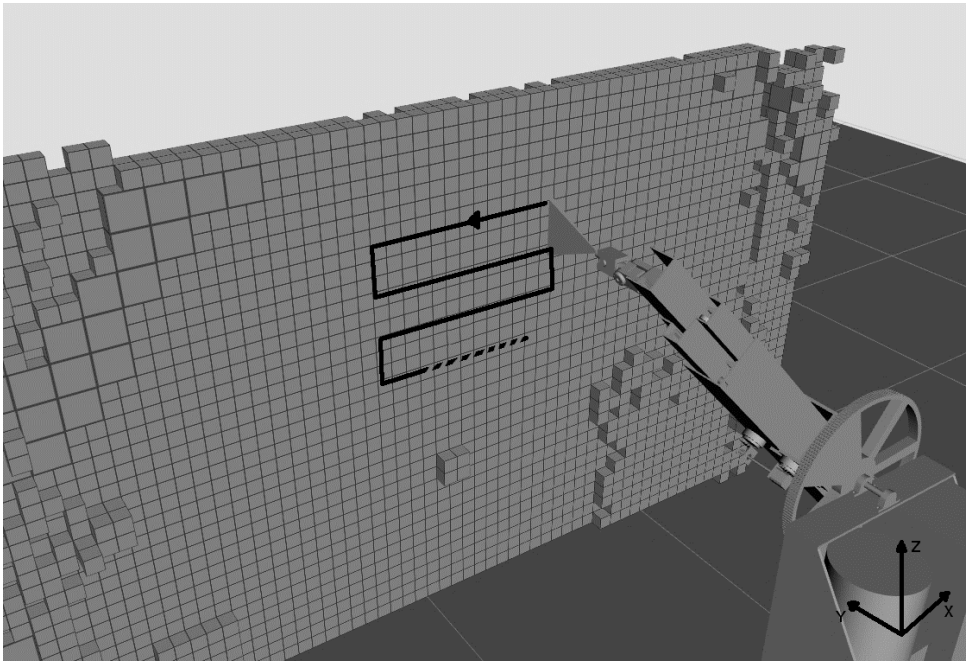
Om octomap in te laden moet eerst de COCO-robot worden opgestart.

```
roslaunch moveit_tutorials octomap_cart_int.launch
```

In het vorige commando wordt de demo.launch opgeroepen van de COCO-robot package en wordt een transformatie uitgevoerd van puntenwolk naar de octomap. Ook deze laatste wordt ingeladen. Vervolgens kan de node de Cartesiaanse planner starten.

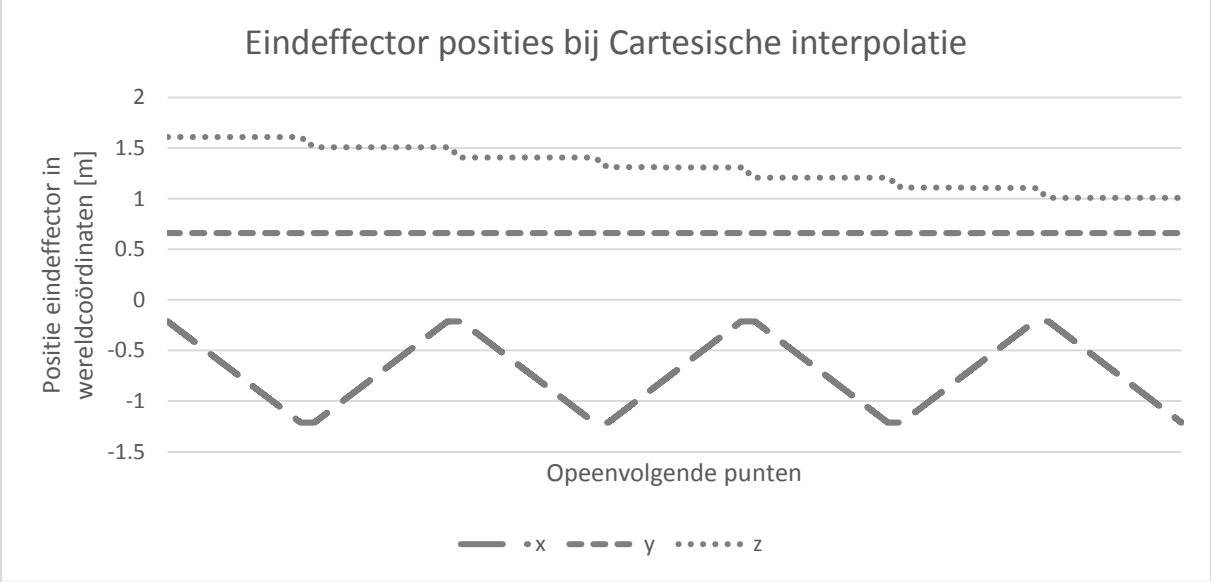
```
rosrun coco_moveit_config moveit_zigzag_waypoints.py
```

Het vorige commando voert het Python bestand uit dat ervoor zorgt dat de planner begint met de zigzagbeweging. In Figuur 34 wordt deze uitgevoerd.



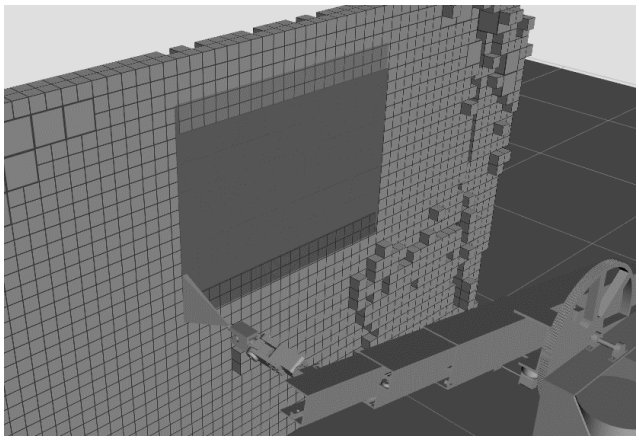
Figuur 34: Omgeving waarin de zigzagbeweging wordt uitgevoerd met Cartesische interpolatie.

Tijdens het plannen worden alle jointposities, alsook jointsnelheden en acceleraties weggeschreven van elk tussenpunt. De positie van de eindeffector wordt berekend met voorwaartse kinematica omdat uit de Cartesische interpolatie joint waarden komen. De zigzagbeweging wordt geplot in Figuur 35, dit is de eindeffectorpositie in xyz-coördinaten.



Figuur 35: Posities van de eindeffector bij Cartesische interpolatie.

In Figuur 35 voert de Cartesische interpolatie een perfecte zigzagbeweging uit. Dit zal dan ook de baseline worden voor alle andere planners. Omdat de zigzagbeweging relatief eenvoudig is, kan de Cartesische interpolatie hier goed mee overweg. Het geverfde gebied in Figuur 36 is gesimuleerd met de posities van Figuur 35.



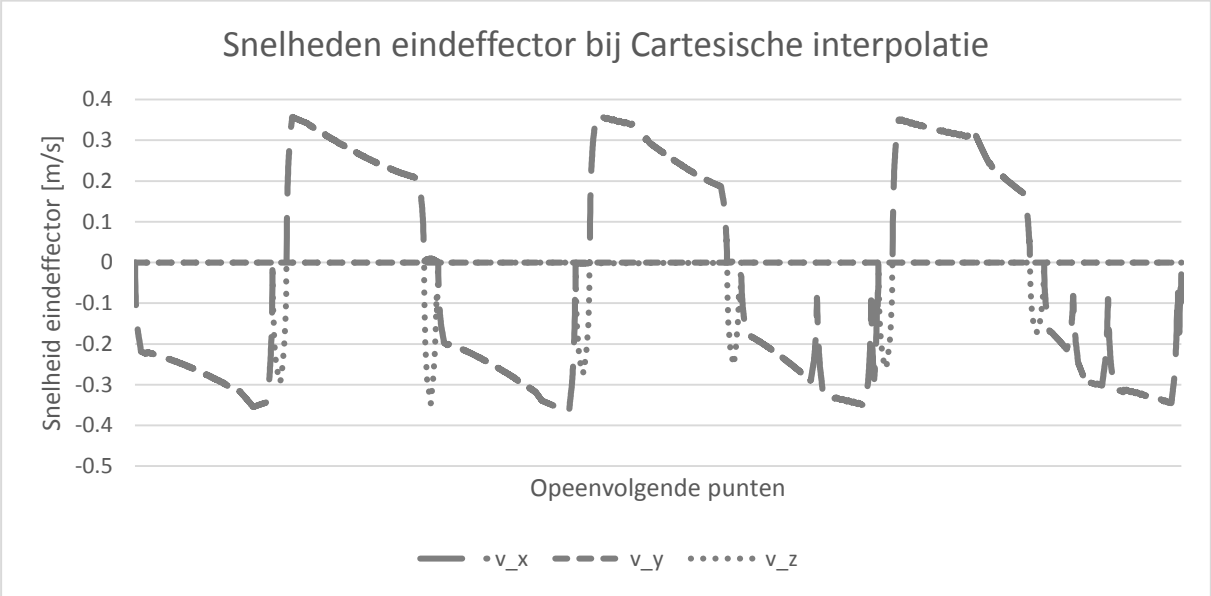
Figuur 36: Het geverfde gebied bij Cartesische interpolatie.

Een vereiste voor de verfrobot is dat de eindeffector met een constante snelheid beweegt eens de spuitkop verf spuit aan een constant debiet. De planner heeft de jointposities, alsook jointsnelheden en acceleraties per configuratie weggeschreven/opgeslagen. Om de jointsnelheden om te vormen naar eindeffectorsnelheden maakt het programma gebruik van een Jacobiaan. Deze is een transformatiematrix tussen jointsnelheden en eindeffectorsnelheden. De Jacobiaan kan berekend

worden met de jointposities van elk punt. De jointsnelheden kunnen dan ingevuld worden in de formule, weergegeven in (1).

$$\begin{Bmatrix} Vx \\ Vy \\ Vz \\ Wx \\ Wy \\ Wz \end{Bmatrix} = Jacobiaan(jointpositie) * \begin{Bmatrix} Va1 \\ Va2 \\ Va3 \\ Va4 \\ Va5 \\ Va6 \\ Va7 \\ Va8 \end{Bmatrix} \qquad (1)$$

Omdat de COCO-robot 8 vrijheidsgraden heeft en er 6 parameters te berekenen zijn heeft deze Jacobiaanmatrix de vorm van 6x8. Voor elke configuratie in het traject wordt een nieuwe Jacobiaan berekend, rekening houdend met de kinematica van de verfrobot. Als dan in (1) de jointsnelheden worden ingevuld kunnen de 6 parameters voor de eindeffector berekend worden. Van deze parameters worden enkel de eerste 3 gebruikt. Het snelheidsprofiel wordt geplot in Figuur 37.



Figuur 37: Snelheden van de eindeffector bij Cartesische interpolatie.

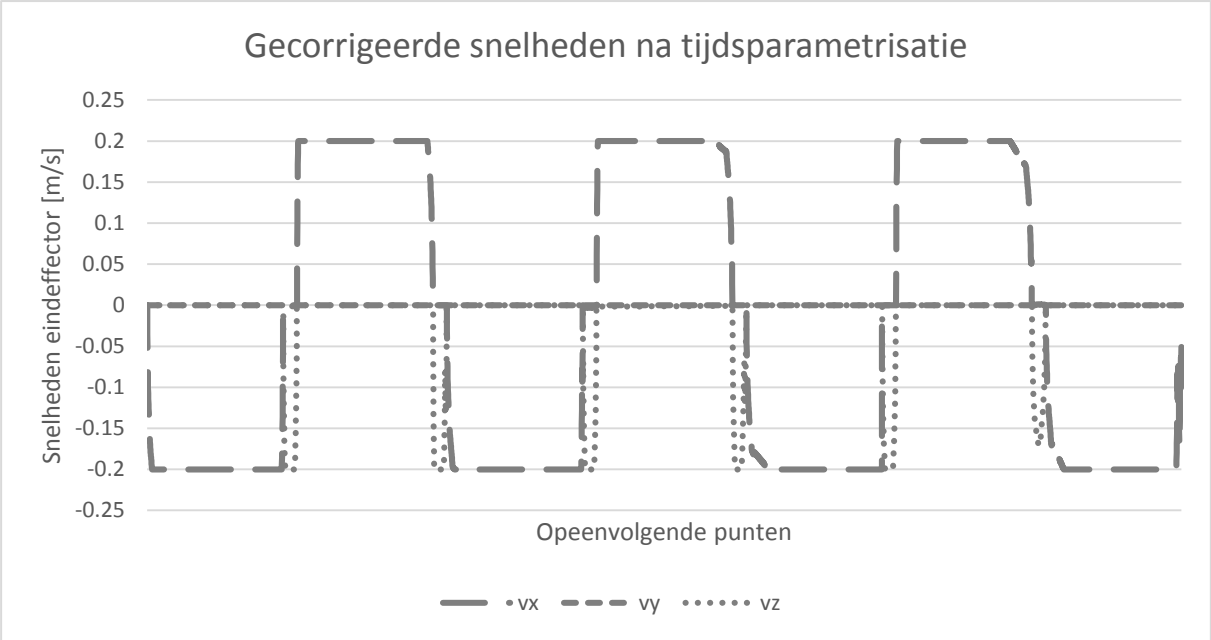
Om te controleren of deze snelheden via de berekening met de Jacobiaan in Figuur 37 wel correct zijn, berekent het programma ook via een alternatieve manier de snelheden van de eindeffector. Tijdens het simuleren schrijft de planner ook het tijdsverschil tussen elke configuratie weg. Op deze manier kan het tijdsverschil tussen relatieve xyz-coördinaten gebruikt worden om xyz-snelheden te berekenen. Deze komen overeen met de berekende snelheden via de Jacobiaanberekening. In Figuur 37 heeft de eindeffector geen constante snelheid. Dit is echter wel een vereiste voor de verftoepassing.

Om toch een aanvaardbaar traject te krijgen is er naar een oplossing gezocht. In dit deel van de masterproef wordt de tijdsparametrisatie-methode gebruikt. Dit betekent kortweg dat het tijdsverschil tussen elke configuratie wordt aangepast om zo de snelheid een constante waarde te geven. Het programma begrenst de eindeffectorsnelheid op 0,2 m/s en past het tijdsverschil tussen elke configuratie aan. In de code wordt dit gedaan met volgende formule weergegeven in Figuur 38.

```
if ((vx >= ingestelde_vx) || (vx <= -ingestelde_vx)){
  dt = abs( dx/ingestelde_vx );
  vx = dx/dt;
}
```

Figuur 38: Formule om tijdsparametrisatie te implementeren.

In Figuur 39 worden de nieuwe snelheden weergegeven.



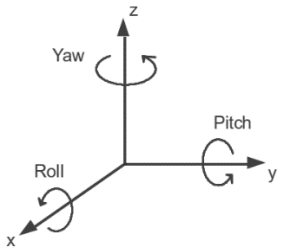
Figuur 39: Nieuwe snelheden van de eindeffector na tijdsparametrisatie.

In Figuur 39 is de eindeffectorsnelheid in de x-richting nu wel constant. De snelheid in de y-richting is nog altijd nul. De snelheid in de z-richting is ook begrensd op 0,2 m/s, maar is niet relevant in deze toepassing. Dit is wanneer de verfspuit verticaal beweegt. De snelheid in de x-richting moet constant zijn, tijdens het verven van het traject, om de verf gelijkmatig te verdelen. Enkel het tijdsverschil wordt aangepast tussen de opeenvolgende punten. De posities blijven dus onveranderd. Het gevolgde traject blijft dus hetzelfde.

De conclusie is dat lineaire interpolatie uitstekend werkt qua berekening van posities. Deze worden goed nagestreefd, maar de bekomen snelheden zijn origineel niet bruikbaar om te verven. Na tijdsparametrisatie is de snelheid van de eindeffector wel geschikt. Tijdens het plannen is ook ondervonden dat de planner niet kan werken met obstakeldetectie. Als het botst met een obstakel zal de Cartesische interpolatie stoppen met berekenen.

5.2 Point-to-point planning met RRTConnect

De RRTConnect planner gebruikt de bibliotheek OMPL om van configuratie A naar B te bewegen. Met behulp van een Python programma wordt de eindeffector van de verfrobot in xyz- en rpy-coördinaten naar zijn einddoel gestuurd. In Figuur 40 wordt weergegeven hoe de rotatie Roll, Pitch en Yaw werken voor een eindeffector.



Figuur 40: Grafische voorstelling van Roll, Pitch en Yaw [29].

Bij de Cartesische interpolatie werd gebruik gemaakt van tussenpunten, maar dat is hier niet het geval. In deze case worden dus enkel point-to-point bewegingen gerealiseerd, geen contourbewegingen.

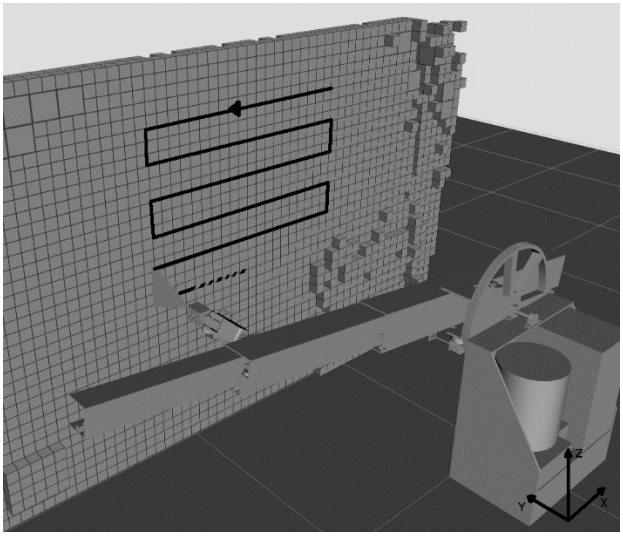
Ook hier moet eerst de octomap worden ingeladen samen met de COCO-robot.

```
roslaunch moveit_tutorials octomap_ompl.launch
```

Vervolgens kan de node worden gestart om de zigzagbeweging uit te voeren.

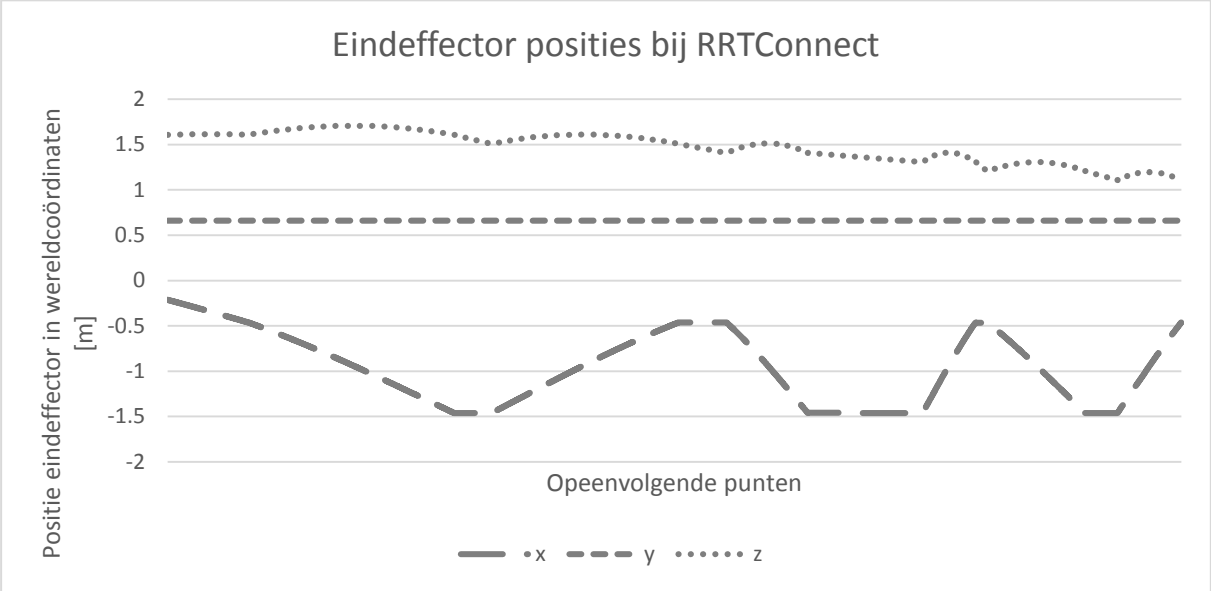
```
rosrun coco_moveit_config moveit_zigzag_pose_goal.py
```

Het vorige commando voert het Python bestand uit dat ervoor zorgt dat de planner begint met de zigzagbeweging. In Figuur 41 wordt deze uitgevoerd.



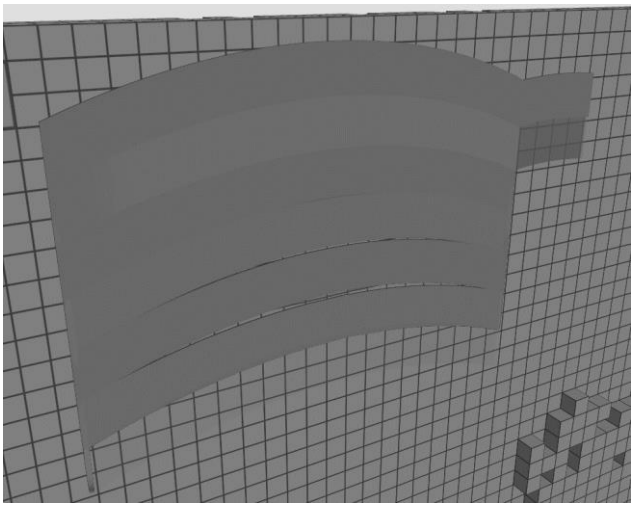
Figuur 41: Omgeving waarin de zigzagbeweging wordt uitgevoerd met RRTConnect.

In Figuur 41 wordt de zigzagbeweging uitgevoerd door RRTConnect. De omgeving is identiek als in Figuur 34. Tijdens het plannen worden alle jointposities, alsook jointsnelheden en acceleraties weggeschreven van het gehele traject. Ook hier worden de posities van de eindeffector berekend met voorwaartse kinematica. Hiervoor werd een cpp-programma geschreven dat deze berekening uitvoert. De zigzagbeweging wordt geplot in Figuur 42, dit is de eindeffectorpositie in xyz-coördinaten.



Figuur 42: Posities van de eindeffector bij RRTConnect.

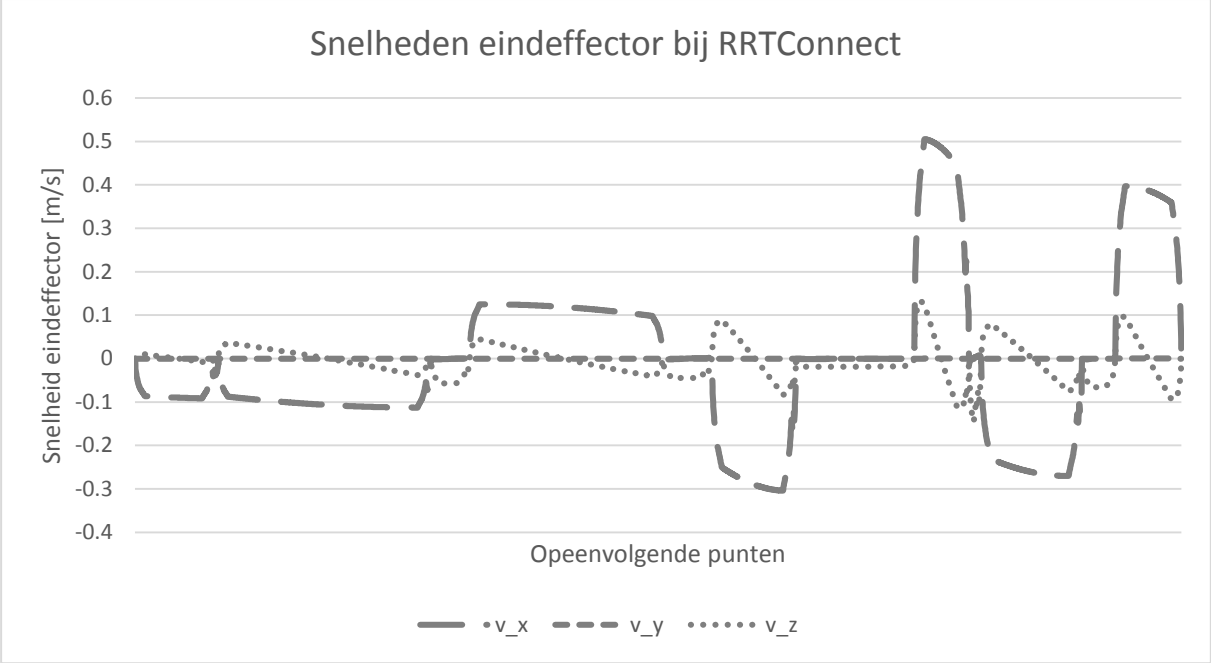
In Figuur 42 voert de planner een minder goede zigzagbeweging uit. Het geverfde gebied in Figuur 43 is gesimuleerd met de posities van Figuur 42.



Figuur 43: Het geverfde gebied bij RRTConnect.

Het geverfde gebied uit Figuur 43 is niet rechtlijnig. Er zit een boogje in elke beweging. Het geverfde gebied is dus niet goed genoeg om aanvaardbaar te zijn voor deze toepassing. De zigzagbeweging wordt hier niet voldoende nauwkeurig uitgevoerd.

Ook in dit deelhoofdstuk wordt de snelheid onderzocht. Ook hier wordt gebruik gemaakt van de Jacobiaan om de jointsnelheden om te vormen tot eindeffectorsnelheden.
Het snelheidsprofiel wordt geplot in Figuur 44.



Figuur 44: Snelheden van de eindeffector bij RRTConnect.

In Figuur 44 heeft de eindeffector geen constante snelheid. Dit is echter wel een vereiste voor de verftoepassing.

De conclusie is dat RRTConnect uit OMPL goed werkt in deze case. Er wordt altijd een pad gevonden van begin- tot eindconfiguratie in deze omgeving. De standaardparameters in de package worden dan ook niet aangepast. De posities worden goed nagestreefd, alleen is het verfgebied niet rechtlijnig. Bij de Cartesische interpolatie methode was het verfgebied wel rechtlijnig. De snelheden van de eindeffector zijn niet constant, dus ook niet bruikbaar voor het verven.

Deze planner werkt wel met obstakeldetectie. RRTConnect heeft een aantal seconden de tijd om een pad te vinden, als dat niet lukt, stopt de berekening. Sampling gebaseerde planners kunnen niet met 100% zekerheid feedback geven of er effectief geen pad mogelijk is. Tevens is het zo dat iedere keer wanneer er een nieuwe zigzagbeweging wordt uitgevoerd, een verschillend pad wordt gecreëerd. Iedere keer is er dus een ander pad, wat minder interessant is voor deze toepassing.

5.3 Point-to-point planning met STOMP

De STOMP planner is een optimalisatie gebaseerde planner. Ook hier wordt een Python programma gebruikt om de eindeffector van de verfrobot in xyz- en rpy-coördinaten naar zijn einddoel te sturen.

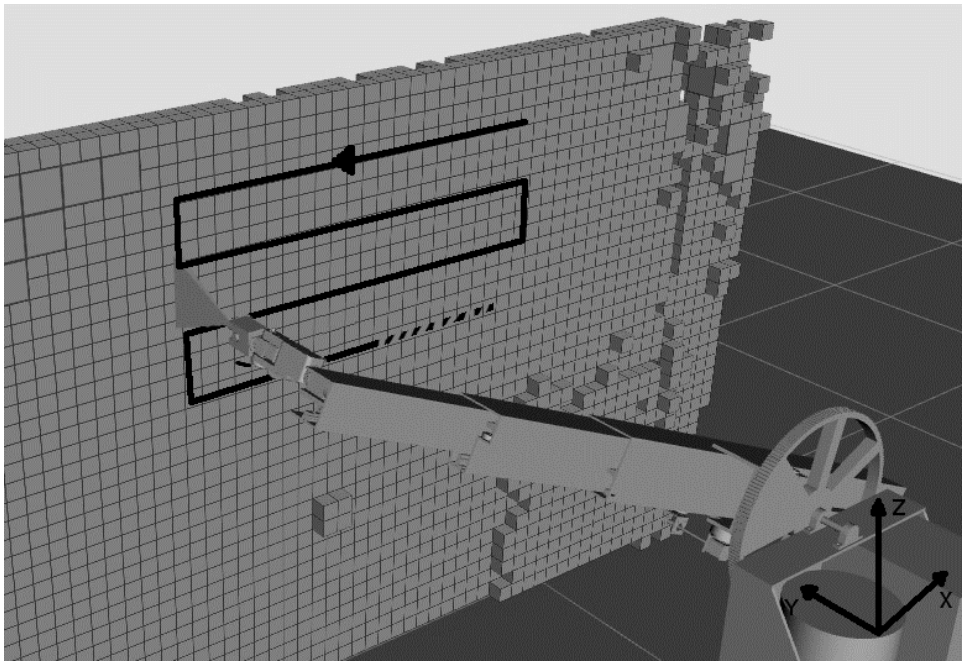
Ook hier moet eerst de octomap worden ingeladen samen met de COCO-robot:

```
roslaunch moveit_tutorials octomap_stomp.launch
```

Vervolgens kan de node worden gestart om de zigzagbeweging uit te voeren:

```
rosrun coco_moveit_config moveit_zigzag_pose_goal.py
```

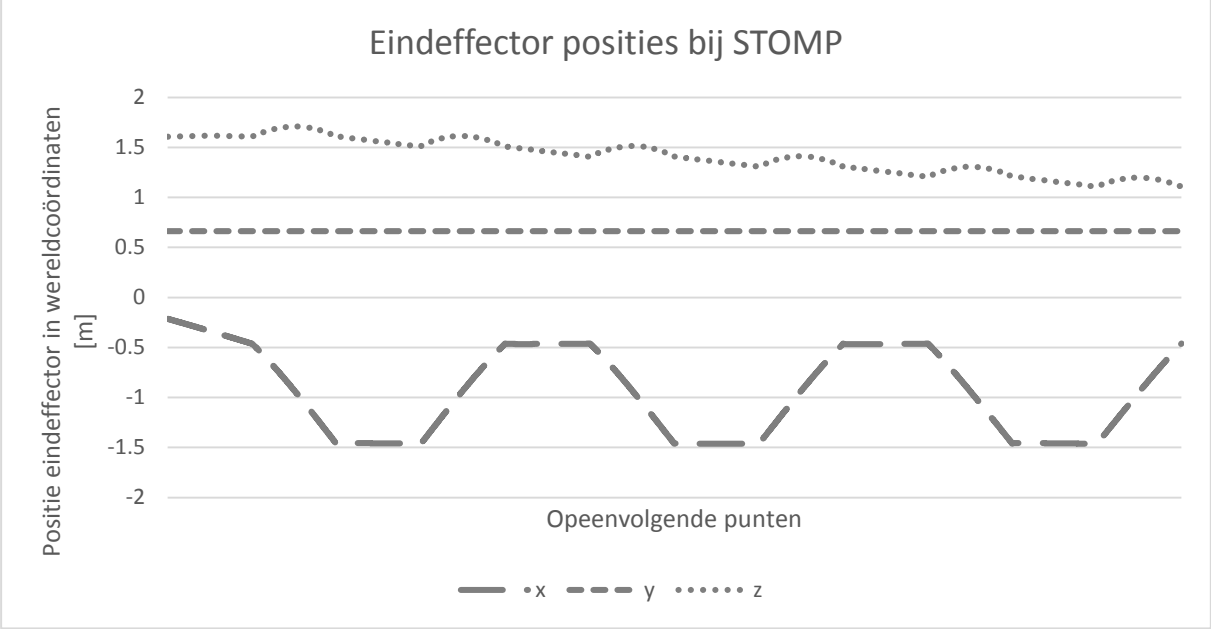
Het vorige commando voert het Python bestand uit dat ervoor zorgt dat de planner begint met de zigzagbeweging. In Figuur 45 wordt deze uitgevoerd.



Figuur 45: Omgeving waarin de zigzagbeweging wordt uitgevoerd met STOMP.

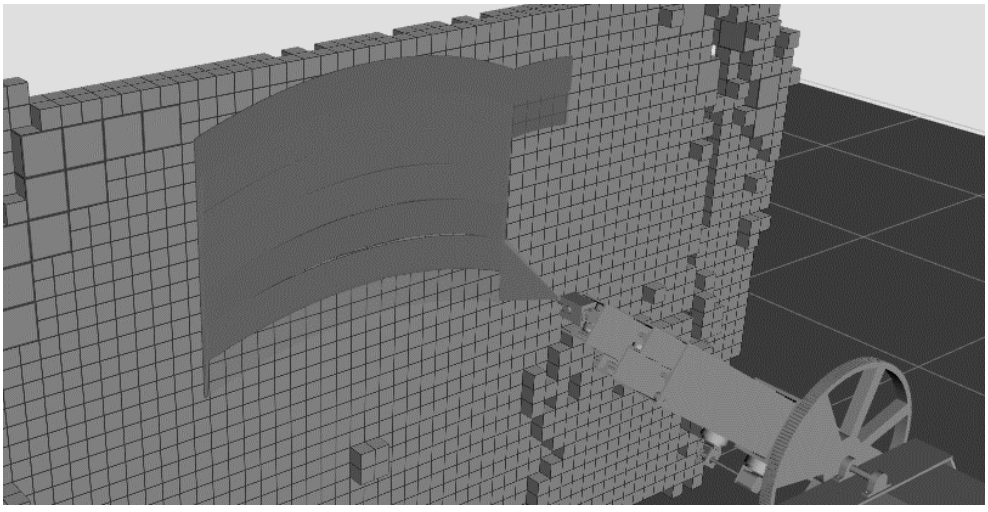
In Figuur 45 wordt de zigzagbeweging uitgevoerd door STOMP. De omgeving is identiek als in Figuur 34.

Tijdens het plannen worden alle jointposities, alsook jointsnelheden en -acceleraties weggeschreven van het gehele traject. Ook hier worden de posities van de eindeffector berekend met voorwaartse kinematica. De zigzagbeweging wordt geplot in Figuur 46, dit is de eindeffectorpositie in xyz-coördinaten.



Figuur 46: Posities van de eindeffector bij STOMP.

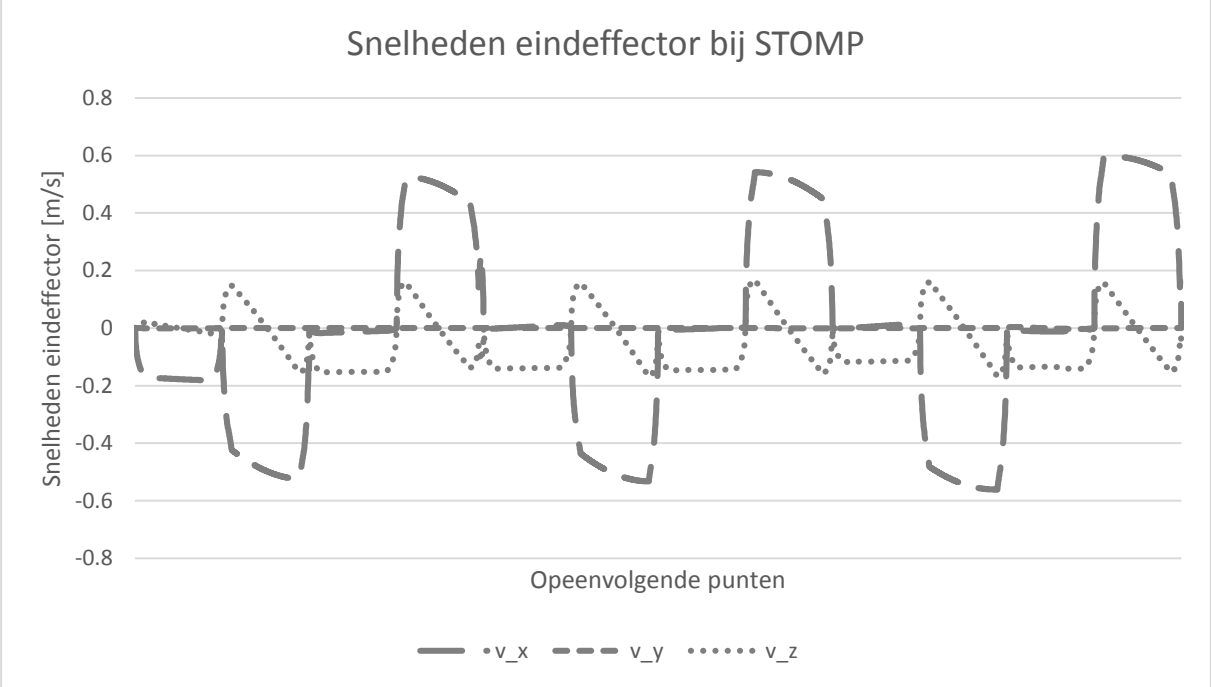
In Figuur 46 voert de planner een minder goede zigzagbeweging uit. Het geverfde gebied in Figuur 47 is gesimuleerd met de posities van Figuur 46.



Figuur 47: Het geverfde gebied bij STOMP.

Het geverfde gebied uit Figuur 47 is ook hier niet rechtlijnig. Er zit telkens een boogje in elke beweging. Het geverfde gebied is dus niet goed genoeg om aanvaardbaar te zijn voor deze toepassing. De zigzagbeweging wordt hier niet voldoende nauwkeurig uitgevoerd.

Ook in dit deelhoofdstuk wordt de snelheid onderzocht door gebruik te maken van de Jacobiaan om de jointsnelheden om te vormen tot eindeffectorsnelheden. Het snelheidsprofiel wordt geplot in Figuur 48.



Figuur 48: Snelheden van de eindeffector bij STOMP.

In Figuur 48 heeft de eindeffector geen constante snelheid. Dit is echter wel een vereiste voor de verftoepassing.

De conclusie is dat STOMP goed werkt in deze case. Er wordt altijd een pad gevonden van begin- tot eindconfiguratie in deze omgeving. De standaardparameters in de package werden wel aangepast, omdat de originele waardes niet goed werken met obstakels. De aangepaste waardes worden getoond in Figuur 49.

```
stomp/endeffector:
  group_name: endeffector
  optimization:
    num_timesteps: 400
    num_iterations: 4000
    num_iterations_after_valid: 0
    num_rollouts: 10
    max_rollouts: 10
    initialization_method: 1 #[1 : LINEAR_INTERPOLATION, 2 : CUBIC_POLYNOMIAL, 3 : MININUM_CONTROL_COST
    control_cost_weight: 0.0
  task:
    noise_generator:
      - class: stomp_moveit/GoalGuidedMultivariateGaussian
        stddev: [0.001, 0.001, 0.001, 0.001, 0.001, 0.001, 0.001, 0.001]
```

Figuur 49: Aangepaste parameterwaardes voor STOMP planner.

De voornaamste veranderde parameters zijn de stddev-parameters. Dit is de hoeveelheid van ruis die op de interpolatie van het pad wordt toegelaten per joint van de robot. In dit geval 8 waardes voor 8 joints van de verfrobot. Door deze waardes aan te passen is het mogelijk om beter obstakels te vermijden.

De posities worden goed nagestreefd, alleen zijn deze niet zo rechtlijnig als bij de Cartesische interpolatie. De snelheden van de eindeffector zijn niet constant, dus ook niet bruikbaar voor het verven.

Deze planner werkt met obstakeldetectie. STOMP heeft een maximaal aantal iteraties om een pad te vinden, als dat niet lukt, stopt de berekening.

5.4 Point-to-point planning met CHOMP

De CHOMP planner is ook een optimalisatie gebaseerde planner. Ook hier wordt een Python programma gebruikt om de eindeffector van de verfrobot in xyz- en rpy-coördinaten naar zijn einddoel te sturen.

Ook hier moet eerst de octomap worden ingeladen samen met de COCO-robot:

```
roslaunch moveit_tutorials octomap_chomp.launch
```

Vervolgens kan de node worden gestart om de zigzagbeweging uit te voeren:

```
rosrun coco_moveit_config moveit_zigzag_pose_goal.py
```

Het vorige commando voert het Python bestand uit dat ervoor zorgt dat de planner begint met de zigzagbeweging. De zigzagbeweging kan niet worden uitgevoerd omdat CHOMP niet kan werken met pose goals. De terminal geeft aan dat de planner enkel met jointgoals kan werken.

Het Python bestand stuurt xyz- en rpy-coördinaten door naar de planner, maar deze kan er dus niet mee overweg. Dit komt omdat CHOMP geen inverse kinematica heeft ingebouwd. Het kan dus niet terugrekenen van positie en oriëntatie in het eindeffector assenstelsel naar joint posities voor de robot. CHOMP kan echter wel werken met joint posities. De 8 joints/vrijheidsgraden aansturen gaat wel, maar dat is voor de toepassing van deze zigzagbeweging niet geschikt. De parameters van CHOMP werden in deze case dan ook niet aangepast.

5.5 Point-to-point planning met TrajOpt

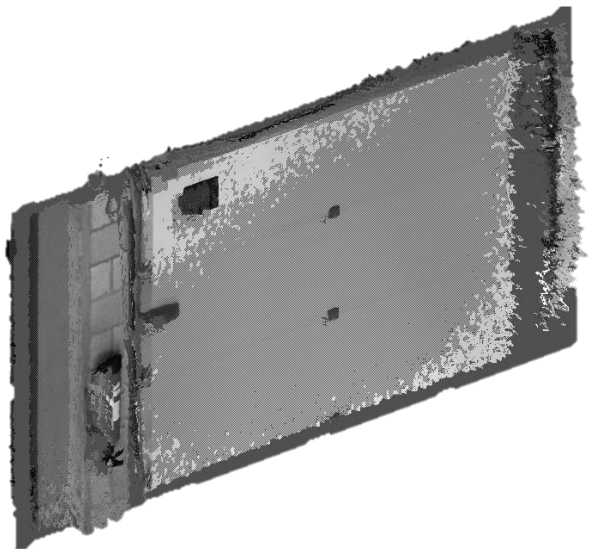
TrajOpt is ook een optimalisatie gebaseerde planner. De integratie met ROS is nieuw en wordt continu bijgewerkt. Er is momenteel nog geen documentatie te verkrijgen, wat het schrijven van code moeilijk maakt. Om TrajOpt te laten werken op ROS is er nood aan een andere package, namelijk "Tesseract" [5]. Deze package zorgt ervoor dat TrajOpt bruikbaar is binnenin RVIZ. In deze masterproef werd TrajOpt getest, omdat deze planner veelbelovend is, beschreven in 2.3.4.

Hier wordt geen Python programma gebruikt, in tegenstelling tot de andere planners. Hier werd de code, de berekening voor het pad, geschreven in Cpp. Het programma berekent de zigzagbeweging door middel van xyz- en rpy- coördinaten en waypoints/tussenpunten in de vorm van constraints.

In deze case wordt de octomap niet ingeladen. In de beperkte beschikbare documentatie van TrajOpt is er geen vermelding van hoe TrajOpt samen met octomap bruikbaar is. TrajOpt werkt met Tesseract, dus het bag-bestand in combinatie met octomap is hier niet bruikbaar. Als het bag-bestand wordt omgevormd tot een pcd-bestand, kan de puntenwolk wel gebruikt worden met TrajOpt. Deze omvorming kan met de volgende commando's worden uitgevoerd.

```
roscore
rosrun pcl_ros bag_to_pcd <input_file.bag> <topic> <output_directory>
```

Bovenstaande commando's werden al besproken in 4.3. Na de conversie kan het pcd-bestand ingeladen worden via CloudCompare. Doorheen de puntenwolk kan nu een vlak worden gefit. Dit wordt weergegeven in Figuur 50.



Figuur 50: Puntenwolk/Pointcloud in CloudCompare met een gefit vlak.

Het gefitte vlak in Figuur 50 kan vervolgens als STL-bestand worden geëxporteerd om later als mesh in te laden m.b.v. Tesseract en TrajOpt.

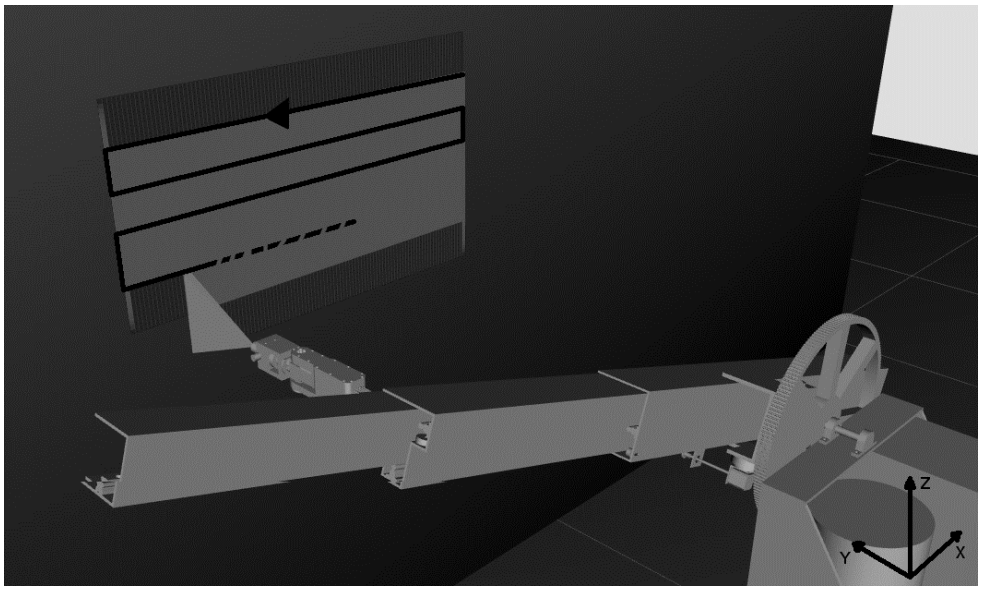
De mesh van de muur kan ingeladen worden samen met de COCO-robot.

```
roslaunch trajopt_examples load_coco_zigzag.launch
```

In deze launch-file wordt automatisch de node (cpp-bestand) gestart om de zigzagbeweging uit te voeren. Deze wordt uitgevoerd met het volgende commando:

```
rosrun trajopt_examples trajopt_examples_coco_zigzag
```

Het vorige commando voert het cpp-bestand uit dat ervoor zorgt dat de planner begint met de zigzagbeweging. In Figuur 51 wordt deze uitgevoerd.

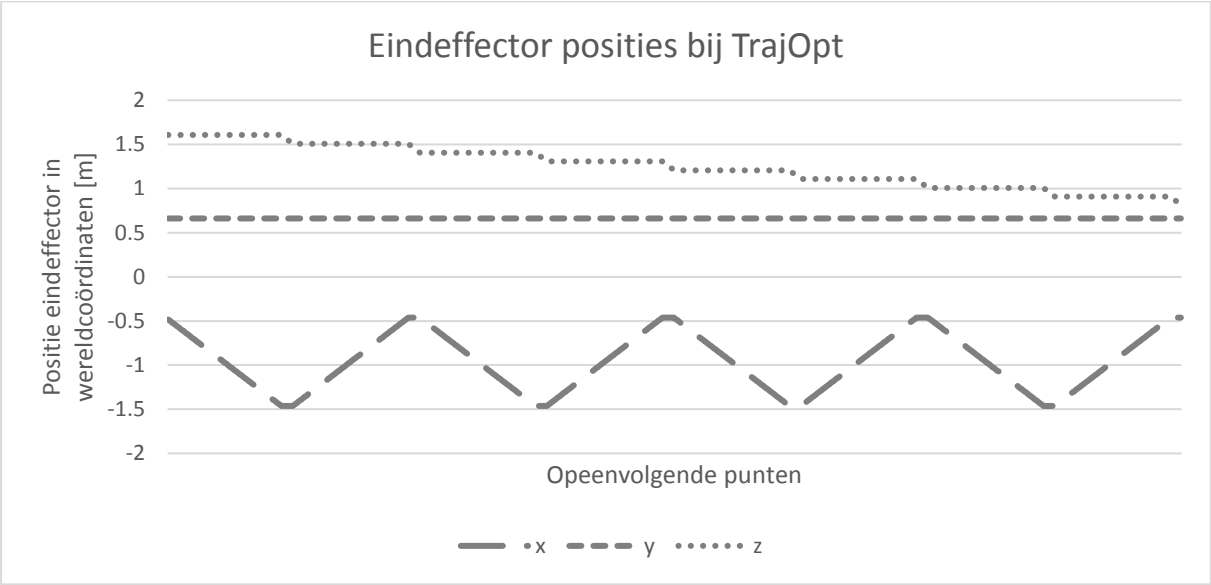


Figuur 51: Omgeving waarin de zigzagbeweging wordt uitgevoerd met TrajOpt.

De waypoints/tussenpunten worden als constraints ingesteld bij TrajOpt. Dit betekent dat voor elk configuratie de xyz- en rpy-coördinaten worden gedefinieerd. In Figuur 51 worden deze tussenpunten perfect gevolgd. Er is geen afwijking mogelijk op deze constraints. De punten moeten worden aangevaren door de robot, wat eigen is aan een constraint definitie (geen tolerantie mogelijk). Tijdens het plannen worden enkel jointposities weggeschreven. Dit komt doordat het traject dat gegenereerd wordt door TrajOpt geen tijdstappen/tijdindicaties heeft.

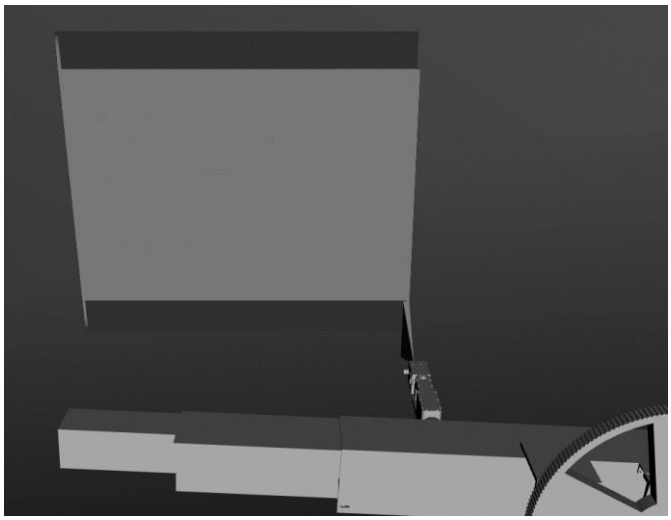
Om toch verder te kunnen rekenen met de gegenereerde posities van TrajOpt wordt gebruik gemaakt van constante tijdsstappen tussen elk punt.

In Figuur 52 worden de posities geplot van de zigzagbeweging.



Figuur 52: Posities van de eindeffector tijdens de zigzagbeweging bij TrajOpt.

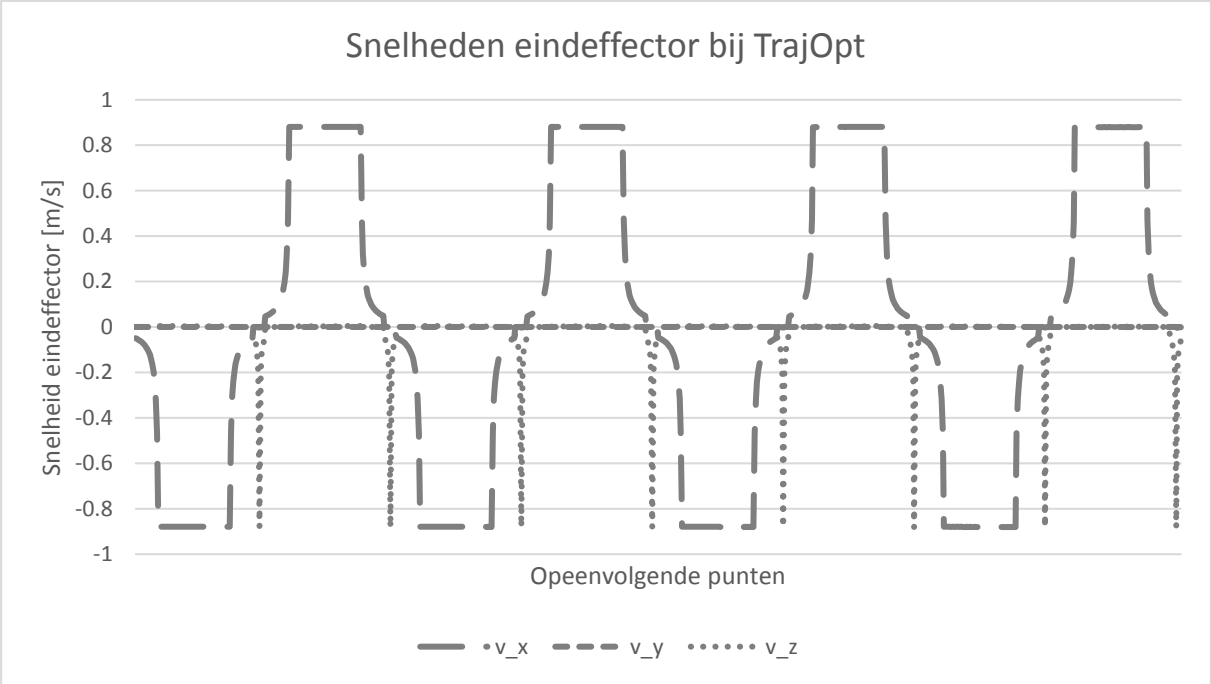
In Figuur 52 voert de planner een zeer goede zigzagbeweging uit. Het geverfde gebied in Figuur 53 is gesimuleerd met de posities van Figuur 52 en is perfect rechtlijnig. De posities bij Cartesische interpolatie en TrajOpt zijn identiek.



Figuur 53: Het geverfde gebied bij TrajOpt.

Het geverfde gebied uit Figuur 53 is wel rechtlijnig. Dit traject is aanvaardbaar voor de verftoepassing. De zigzagbeweging wordt perfect uitgevoerd.

Ook in dit deelhoofdstuk wordt de snelheid onderzocht. Eerder werd vermeld dat TrajOpt enkel posities wegschrijft. Daarom is er nog een node geschreven die, m.b.v. de constante tijdsintervallen, de jointsnelheden en acceleraties wegschrijft. Zo kan ook hier weer gebruik gemaakt worden van de Jacobiaan om de jointsnelheden om te vormen tot eindeffectorsnelheden.
Het snelheidsprofiel wordt geplot in Figuur 54.



Figuur 54: Snelheden van de eindeffector tijdens de zigzagbeweging bij TrajOpt.

In Figuur 54 heeft de eindeffector een constante snelheid, nadat deze is aangelopen tot de maximale waarden. Dit is een goede eigenschap voor de verfbeweging.

De conclusie is dat TrajOpt uitstekend werkt. Er wordt altijd een pad gevonden van begin- tot eindconfiguratie in deze omgeving. Er werden geen parameters veranderd in de package om met de obstakels om te gaan. Verder worden de posities goed nagestreefd en is alles mooi rechtlijnig. In vergelijking met de Cartesische interpolatie is deze planner qua genereren van posities identiek. De snelheid van de eindeffector is constant doordat de code gebruik maakt van een constant tijdsinterval. De aanlooptijd en aflooptijd zijn beide instelbaar, maar werden in deze masterproef niet geïmplementeerd.

Deze planner werkt met obstakeldetectie. TrajOpt geeft aan dat het totale traject (van de zigzagbeweging) obstakelvrij is. Echter wanneer er toch een botsing voorkomt, stelt TrajOpt geen alternatief voor omdat de code gebruik maakt van tussenpunten. Deze punten zijn altijd als constraints gedefinieerd. Voor de planner is dus elke configuratie geforceerd en moet de robot in elk tussenpunt geweest zijn. De punten liggen kort bij elkaar, waardoor het obstakel niet vermeden kan worden. Niettemin is het positief dat TrajOpt aangeeft dat er een botsing gebeurt in het traject. Cartesische interpolatie geeft dit niet aan, wat TrajOpt een voordeel geeft.

Omdat de TrajOpt planner goed werkt in deze case, is in dit deelhoofdstuk ook nog een variant op de case toegevoegd. Een andere toepassing/case voor deze verfrobot zou het verven van een pilaar zijn. Deze zijn vaak hoog en moeilijk bereikbaar met een ladder. Een ideale case voor deze robot, want hier moet ook rekening gehouden worden met de oriëntatie van de spuitkop.

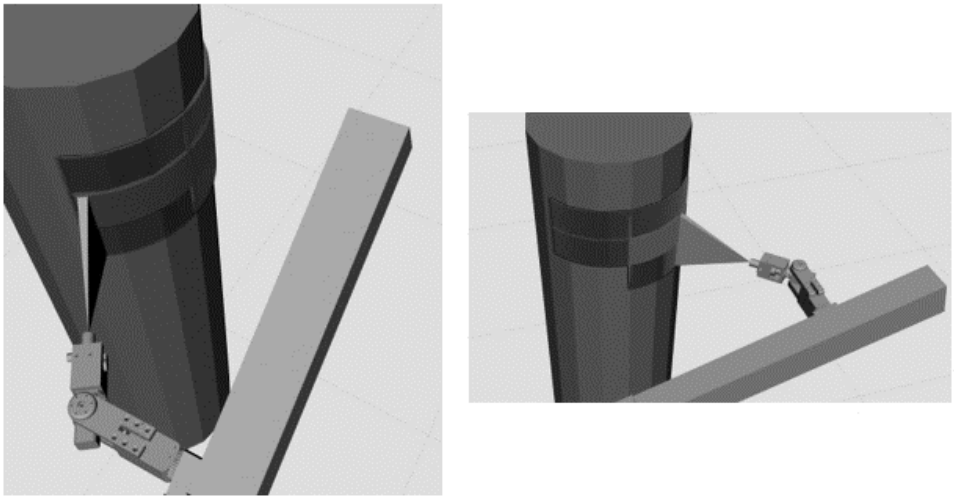
In deze toepassing wordt geen gebruik gemaakt van een gescande omgeving. Enkel een STL-bestand, dat een pilaar voorstelt, wordt ingeladen.

```
roslaunch trajopt_examples load_coco_pilaar.launch
```

In deze launch-file wordt automatisch de node (cpp-bestand) gestart om de pilaar te verven. Deze wordt uitgevoerd met het volgende commando:

```
rosrun trajopt_examples trajopt_examples_coco_pilaar
```

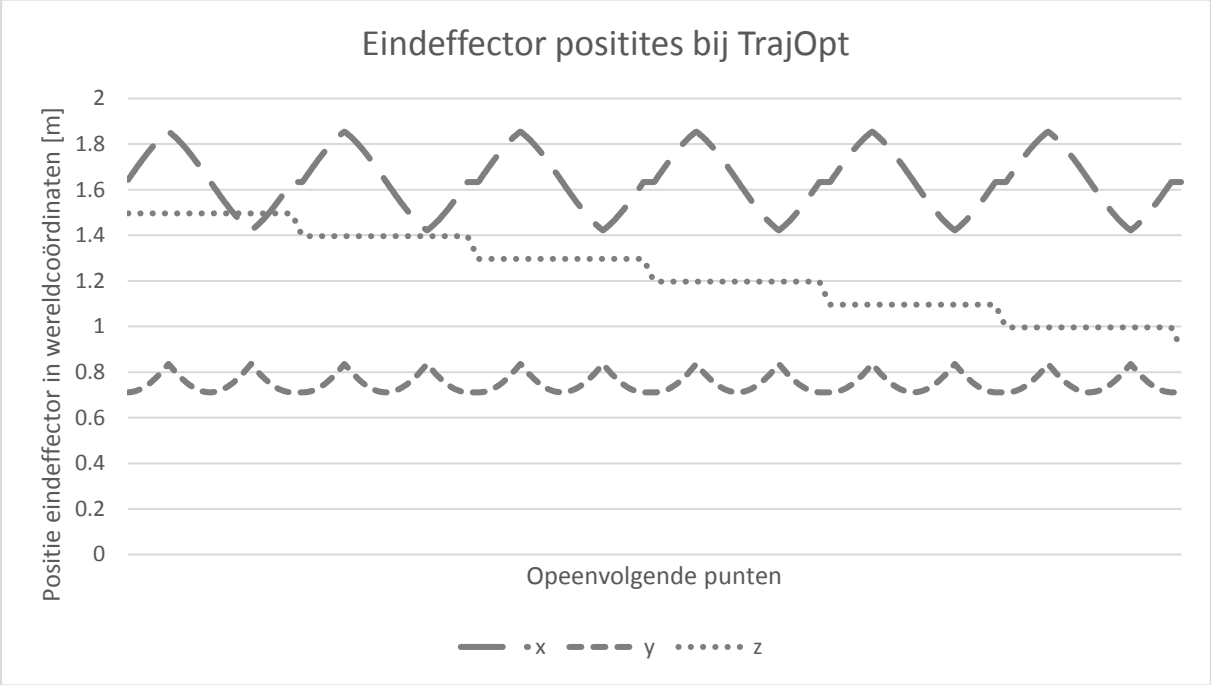
Het vorige commando voert het cpp-bestand uit dat ervoor zorgt dat de planner begint met de pilaarbeweging. In Figuur 55 wordt deze uitgevoerd.



Figuur 55: COCO-robot voert de pilaarbeweging uit met TrajOpt.

Bij deze case is het tevens belangrijk dat de spuitkop ook de juiste oriëntatie per positie aanneemt. Met de constraints van TrajOpt werkt dit goed.

Het gegenereerde traject, om de pilaar te verven, wordt weergegeven in Figuur 56. Dit zijn de eindeffectorposities in xyz-coördinaten.



Figuur 56: Posities van de eindeffector tijdens de pilaarbeweging bij TrajOpt.

Per baan wordt in Figuur 56 een totale hoek van 120° geverfd. Zo kan de verfrobot een volledige omtrek verven door 3x deze beweging uit te voeren op een andere positie. De operator zal de verfrobot manueel moeten verplaatsen om dit te verwezenlijken.

5.6 Conclusie

De verschillende planners werden onderzocht en getest op de verfrobot in dit hoofdstuk. In dit deelhoofdstuk wordt een overzicht aan bevindingen besproken.

Cartesische interpolatie werkt zeer goed tot nagenoeg perfect voor het berekenen van de posities van de eindeffector tijdens de zigzagbeweging. Tussen de begin- en eindconfiguraties worden telkens waypoints berekend met een opgegeven resolutie. Ook de oriëntatie van de spuitkop blijft altijd constant bij de waypoints. De snelheid van de eindeffector is bij deze planner niet constant. Om deze toch constant te krijgen kan de methode van tijdsparametrisatie toegepast worden op het traject. Zo kan deze toch bruikbaar zijn om te verven met de robot. Helaas is er geen obstakeldetectie ingebouwd in deze planner. De berekening van het traject stopt zelfs als deze in botsing komt met de gescande omgeving. De obstakeldetectie is een vereiste voor deze case, want de verfrobot mag nooit in contact komen met de gescande obstakels.

RRTConnect en STOMP kunnen beide xyz-coördinaten aan, maar geen waypoints tussen begin- en eindconfiguratie. De zigzagbeweging is daardoor niet rechtlijnig en ook de oriëntatie is hier vaak niet constant. De spuitkop kan tussen begin- en eindconfiguratie verdraaien omdat deze planners geen tussenpunten definiëren. De snelheid van de eindeffector is bij deze planner ook niet constant. Ook hier kan gebruik worden gemaakt van tijdsparametrisatie, om de snelheid constant te krijgen. Het gegenereerde pad van de twee planners is niet rechtlijnig, dus dit voldoet niet aan de voorwaarde voor deze case. Hier is echter wel obstakeldetectie ingebouwd, de robot zal dus nooit botsen met de gescande omgeving. RRTConnect kan met standaardparameters goed overweg met de obstakels, terwijl STOMP nog wat aanpassingen nodig had. De algemene conclusie is dat RRTConnect en STOMP niet gemaakt zijn om rechtlijnige beweging uit te voeren. Deze academische planners zijn ontwikkeld om de obstakels van de omgeving goed te kunnen vermijden. De conclusie is dat de twee planners niet geschikt zijn voor deze case.

CHOMP kan geen xyz-coördinaten verwerken en dus ook geen waypoints uitvoeren. Deze planner heeft geen inverse kinematica ingebouwd. Enkel jointgoals zijn toegelaten. Deze planner is niet geschikt, want de zigzagbeweging kan niet worden uitgevoerd.

De TrajOpt planner werkt uitstekend. De waypoints/volgpunten worden met xyz-coördinaten als constraints gedefinieerd. De zigzagbeweging is even rechtlijnig als de Cartesische interpolatie methode. Ook de snelheid van de eindeffector is constant, doordat de code gebruik maakt van een constant tijdsinterval. Omdat TrajOpt goed met constraints werkt, werd hier nog een extra case toegevoegd. De pilaarbeweging wordt perfect uitgevoerd, rekening houdend met de oriëntatie van de spuitkop t.o.v. de pilaar.

De conclusie van deze case is dat TrajOpt het meest geschikt is voor de verftoepassing.

6 Evaluatie van de padplanners voor point-to-point bewegingen met de KR5-robot

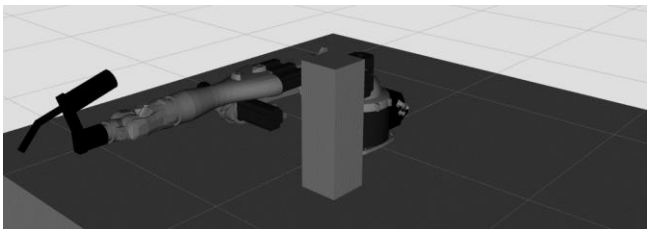
In dit hoofdstuk worden jointgoals vergeleken met de besproken planners uit 2.3. In deze case worden de padplanners besproken voor point-to-point bewegingen, d.w.z. dat de begin- en eindconfiguratie vastligt. Omdat CHOMP niet kan werken met xyz-coördinaten bespreekt dit hoofdstuk de jointgoals. Dit betekent bewegen van configuratie A naar B m.b.v. jointposities/jointhoeken. Tijdens het plannen is het erg belangrijk dat de robot nooit in aanraking komt met obstakels.

Elke planner wordt onderworpen aan een evaluatie in dit hoofdstuk. Het gegenereerde traject moet van configuratie A naar B bewegen rekening houdend met de omgeving. De tussenliggende beweging is minder van belang, zolang de obstakels worden vermeden. Cartesische interpolatie wordt in dit hoofdstuk niet gebruikt, omdat het geplande traject geen lineaire beweging moet afleggen. Elke planner wordt, in dit hoofdstuk, aangestuurd met het volgende Python programma:

```
rosrun kr5_moveit_config moveit_joint_goals.py
```

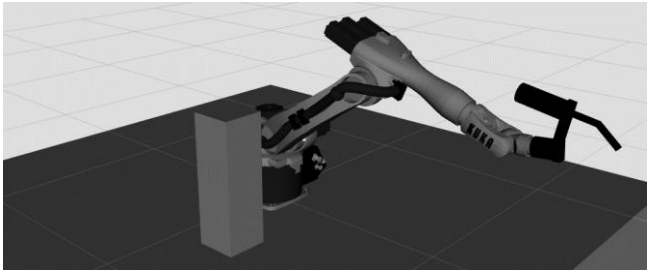
Het vorige commando voert het Python bestand uit wat ervoor zorgt dat de gekozen planner begint. Elke planner wordt op dezelfde manier aangestuurd qua start- en eindconfiguratie. De KR5-robot heeft 6 vrijheidsgraden. Het start- en eindconfiguratie wordt aangestuurd m.b.v. een jointgoal. Dit zijn dus de hoeken per joint uitgedrukt in radialen. Een configuratie wordt voorgesteld als [A1; A2; A3; A4; A5; A6].

De startconfiguratie is ingesteld op [0; 0; 0; 0; 0; 0], weergegeven in Figuur 57.



Figuur 57: De startconfiguratie voor de KR5-robot.

De eindconfiguratie is ingesteld op [-π/2; -0.6; 1.0; 0; -0.5; 0], weergegeven in Figuur 58.



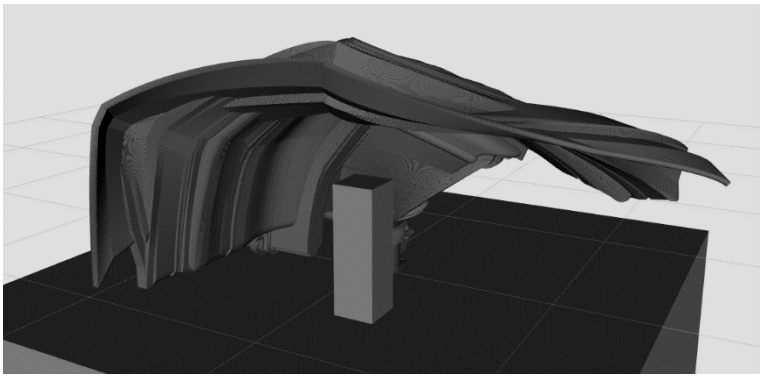
Figuur 58: De eindconfiguratie voor de KR5-robot.

6.1 Point-to-point planning met RRTConnect

De RRTConnect planner gebruikt de bibliotheek OMPL om van configuratie A naar B te bewegen. Om RVIZ met de KR5-robot op te starten wordt volgend commando gebruikt:

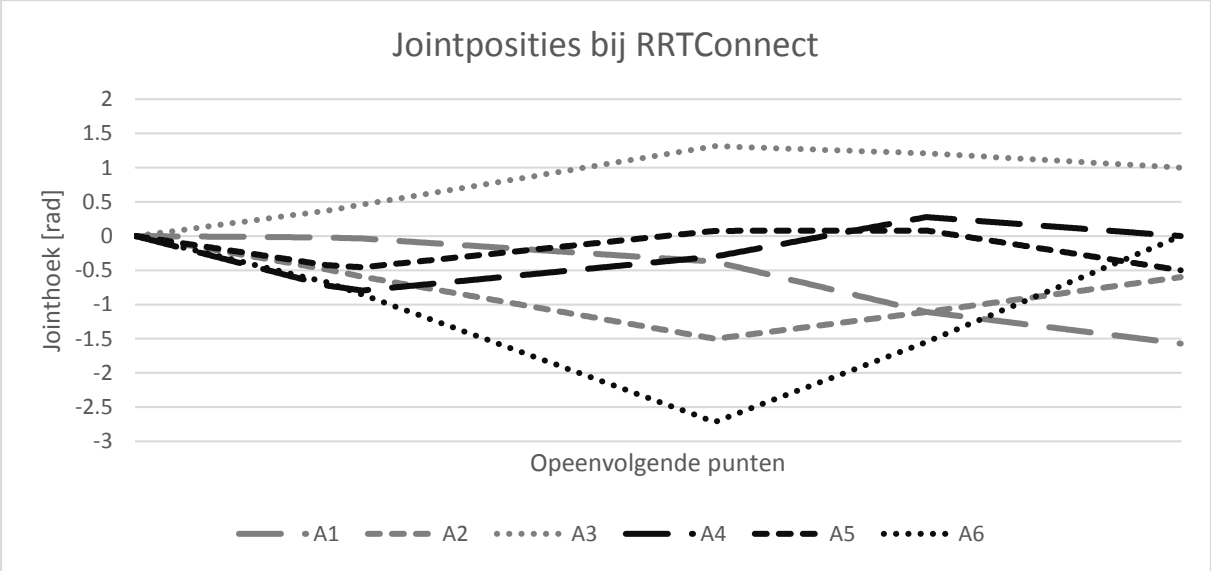
```
roslaunch kr5_moveit_config demo_ompl.launch
```

Vervolgens start ook het Python programma. De planner berekent een pad van start- naar eindconfiguratie, rekening houdend met het obstakel in de omgeving. Dit gegenereerde pad wordt voorgesteld in Figuur 59.



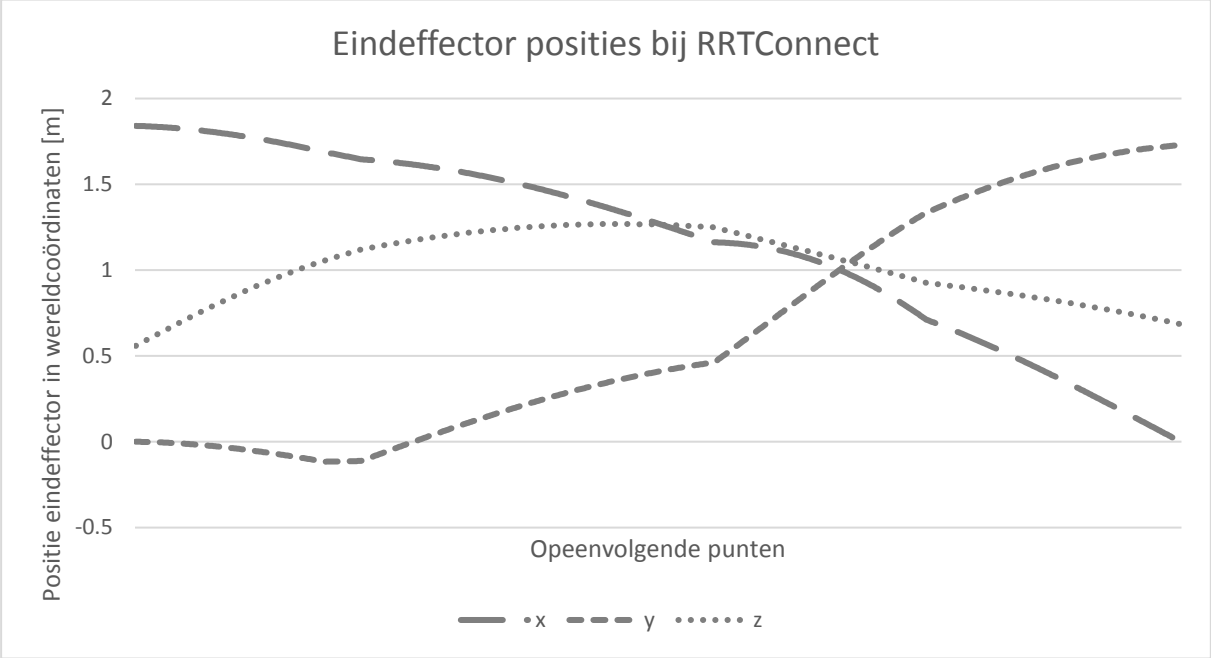
Figuur 59: Het berekende traject bij RRTConnect.

In Figuur 59 voert RRTConnect een correcte beweging uit. Het obstakel in de omgeving wordt vermeden. Tijdens het plannen worden alle jointposities, alsook jointsnelheden en acceleraties weggeschreven van het gehele traject. In Figuur 60 worden de jointposities weergegeven van de berekende baan uit Figuur 59.



Figuur 60: Jointposities bij RRTConnect.

Verder worden hier ook de posities van de eindeffector berekend met voorwaartse kinematica. Deze wordt geplot in Figuur 61, dit is de eindeffectorpositie in xyz-coördinaten.



Figuur 61: Posities van de eindeffector bij RRTConnect.

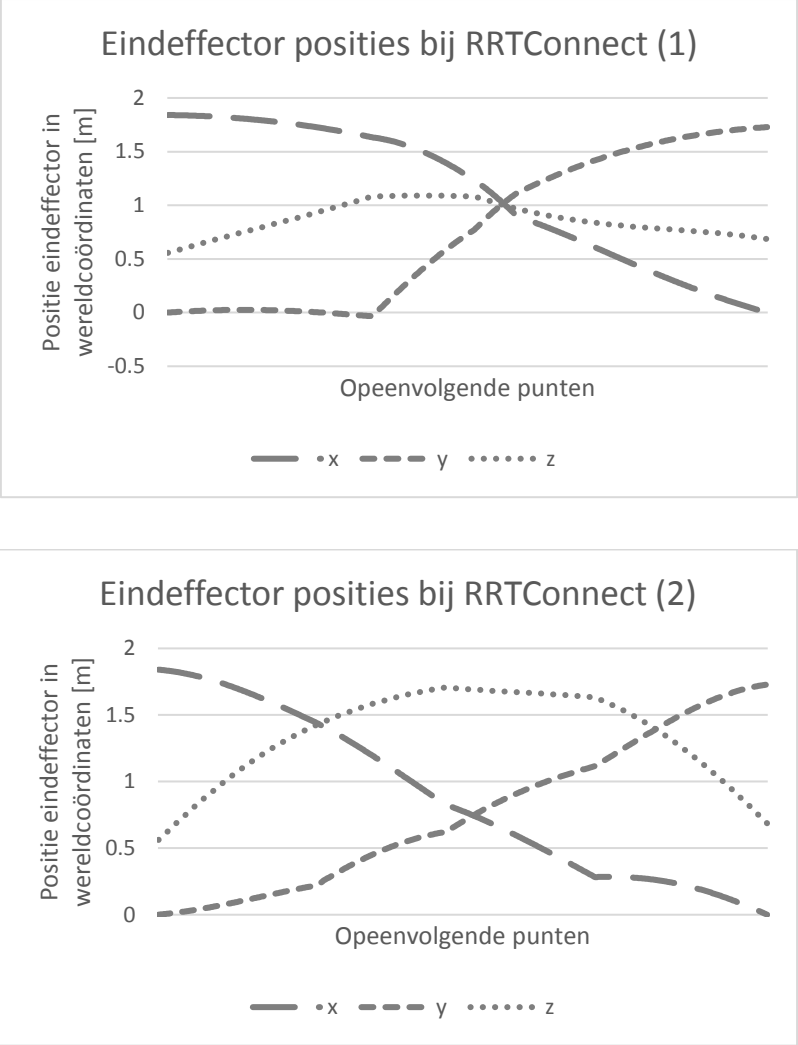
In Figuur 61 worden de posities van de eindeffector weergegeven. Deze komen overeen met het traject uit Figuur 59, waarbij het obstakel in de omgeving wordt vermeden. De planner werkt goed met de standaardparameters, weergegeven in Figuur 62.

```
RRTConnect:
  type: geometric::RRTConnect
  range: 0.0  # Max motion added to tree. ==> maxDistance_ default: 0.0, if 0.0, set on setup()
  longest_valid_segment_fraction: 0.0005
```

Figuur 62: Ingestelde parameters voor RRTConnect.

De parameters in Figuur 62 werden niet aangepast. De parameter “longest valid segment fraction” werd al besproken in 2.3.1.

Het is wel zo dat bij een OMPL planner elke keer een verschillend pad wordt gegenereerd. Het RRTConnect-algoritme vindt altijd een oplossing voor deze case. In Figuur 63 worden twee verschillende trajecten, voor hetzelfde padplanningsprobleem, geplot.



Figuur 63: Twee verschillende trajecten die gegenereerd worden door RRTConnect.

In Figuur 63 worden twee verschillende trajecten gegenereerd. Beide trajecten vermijden het obstakel, ondanks ze toch zoveel verschillen. Iedere keer, wanneer de node wordt uitgevoerd, wordt er dus een ander traject als oplossing voor dit probleem gegenereerd, dit is eigen aan een sampling gebaseerde padplanner.

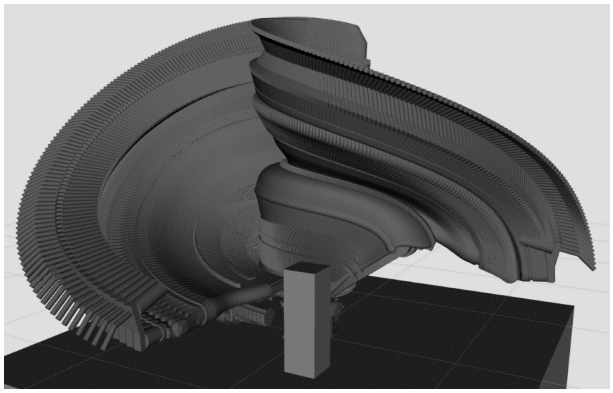
De conclusie is dat RRTConnect uit OMPL goed werkt in deze case. Er werd een pad gevonden van begin- tot eindconfiguratie in deze omgeving. De standaardparameters werden niet aangepast en het obstakel in de omgeving wordt vermeden. Een nadeel is echter dat er iedere keer een ander traject als oplossing wordt gegenereerd. Er zijn knikken in het traject zichtbaar, wat eigen is aan RRTConnect. Deze schokkende bewegingen zorgen voor hoge accelaraties, wat de levensduur van de robot beïnvloed.

6.2 Point-to-point planning met STOMP

Dit deelhoofdstuk evalueert de planner STOMP om van configuratie A naar B te bewegen.
Om RVIZ met de KR5-robot op te starten wordt volgend commando gebruikt:

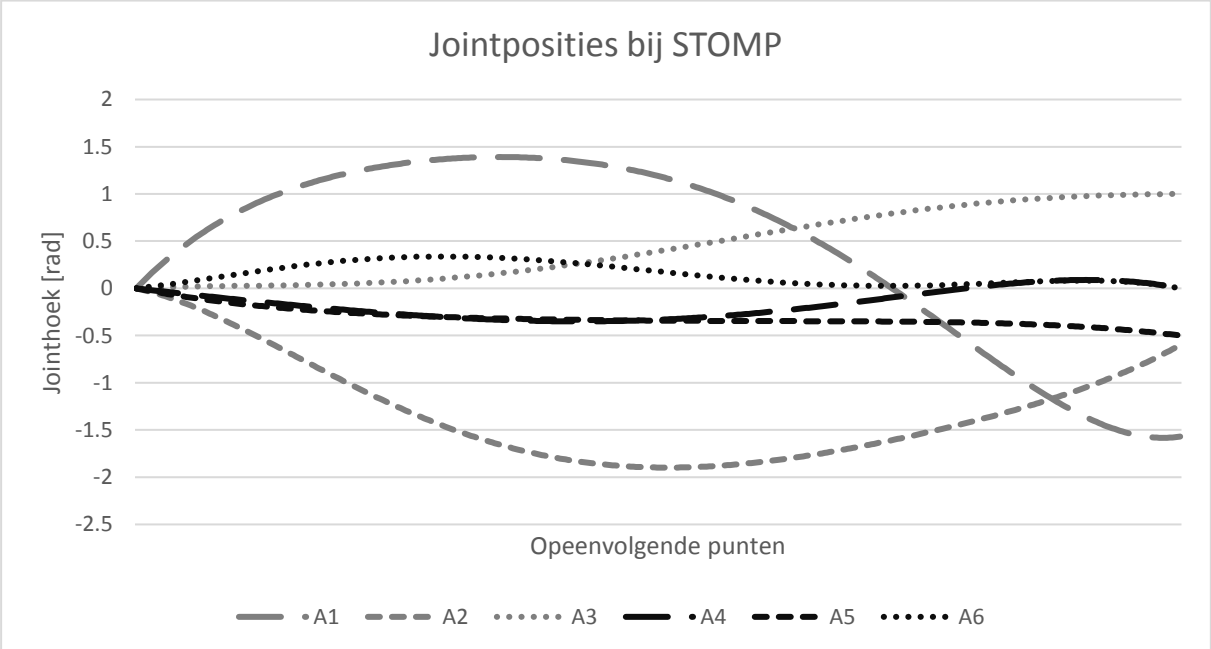
```
roslaunch kr5_moveit_config demo_stomp.launch
```

Vervolgens start ook het Python programma. De planner berekent een pad van start- naar eindconfiguratie, rekening houdend met het obstakel in de omgeving. Dit gegenereerde pad wordt voorgesteld in Figuur 64.



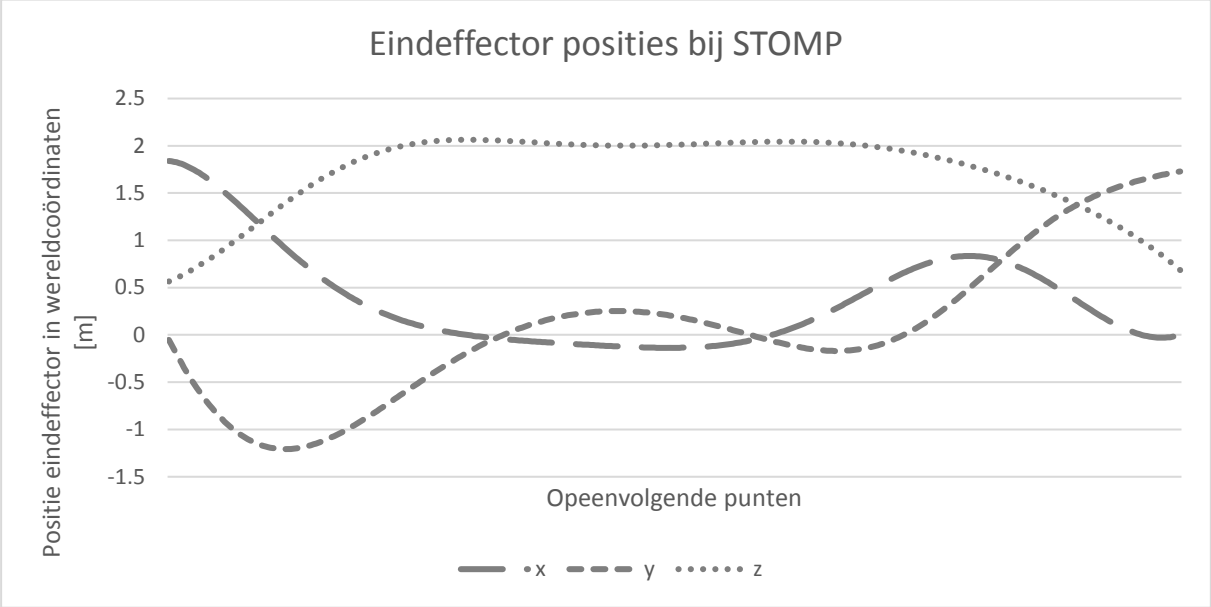
Figuur 64: Het berekende traject bij STOMP.

In Figuur 64 voert STOMP een correcte, vloeiende beweging uit. Het obstakel in de omgeving wordt vermeden. In Figuur 65 worden de jointposities weergegeven van de berekende baan uit Figuur 64.



Figuur 65: Jointposities bij STOMP.

Verder worden hier ook de posities van de eindeffector berekend met voorwaartse kinematica. Deze wordt geplot in Figuur 66, dit is de eindeffectorpositie in xyz-coördinaten.



Figuur 66: Posities van de eindeffector bij STOMP.

In Figuur 66 worden de posities van de eindeffector weergegeven. Deze komen overeen met het traject uit Figuur 64, waarbij het obstakel in de omgeving wordt vermeden. Met de standaardparameters vindt STOMP enkel een traject als er geen obstakels in de omgeving zijn. De parameters werden dus aangepast om toch te kunnen plannen met obstakels, deze worden weergegeven in Figuur 67.

```
stomp/endeffector:
  group_name: endeffector
  optimization:
    num_timesteps: 400
    num_iterations: 4000
    num_iterations_after_valid: 0
    num_rollouts: 10
    max_rollouts: 10
    initialization_method: 1 #[1 : LINEAR_INTERPOLATION, 2 : CUBIC_POLYNOMIAL, 3 : MININUM_CONTROL_COST
    control_cost_weight: 0.0
  task:
    noise_generator:
      - class: stomp_moveit/GoalGuidedMultivariateGaussian
        stddev: [0.25, 0.25, 0.25, 0.1, 0.1, 0.1]
```

Figuur 67: Aangepaste parameters voor STOMP.

De parameter “number of iterations” in Figuur 67 werd verhoogd in waarde. Zo is het toegelaten dat STOMP meer berekeningen maakt om een oplossing voor het padplanningsprobleem te vinden. Verder werd ook de parameter “stddev” verhoogd in waarde. Deze parameter is een maat van ruis op de gewrichten of m.a.w. een grotere waarde komt overeen met een grotere beweging voor het ingestelde gewricht, joint of vrijheidsgraad. Deze parameter werd ook uitgelegd in 2.3.3.

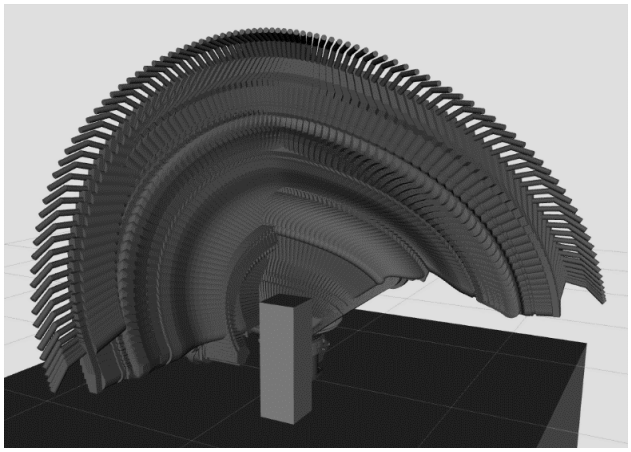
Bij STOMP is het gegenereerde traject, in tegenstelling tot RRTConnect uit OMPL, altijd identiek. De conclusie is dat STOMP goed werkt in deze case, maar het resultaat is niet optimaal. De standaardparameters werden aangepast om met obstakels in de omgeving te kunnen werken. Met de aangepaste parameters wordt het obstakel vermeden.

6.3 Point-to-point planning met CHOMP

Dit deelhoofdstuk evalueert de planner CHOMP om van configuratie A naar B te bewegen.
Om de simulatie met de KR5-robot op te starten wordt volgend commando gebruikt.

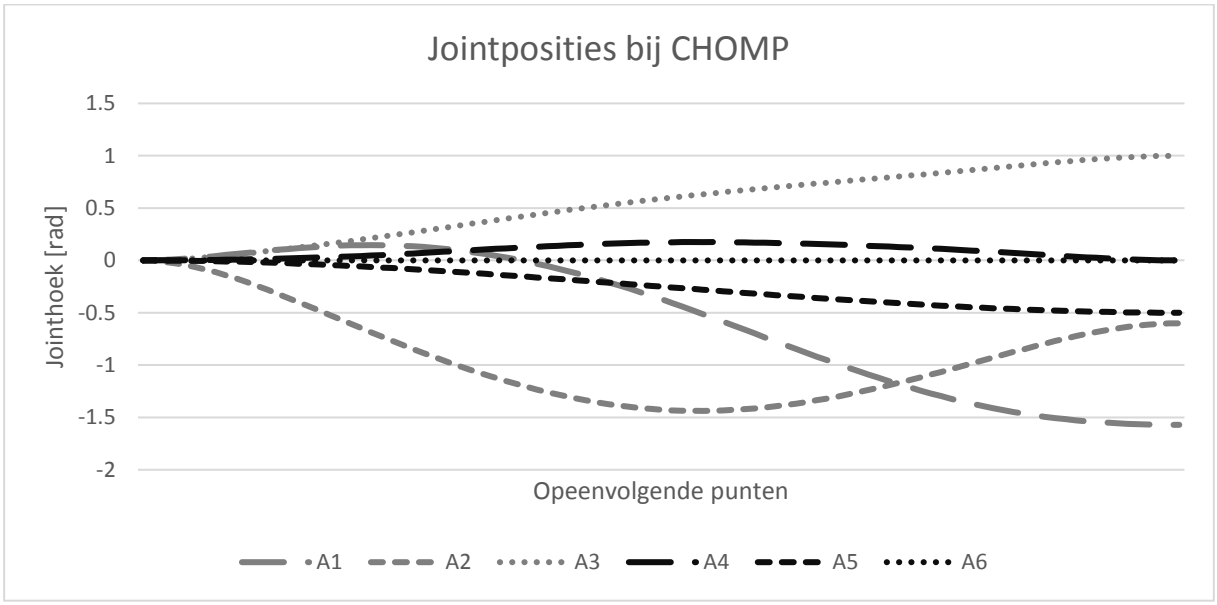
```
roslaunch kr5_moveit_config demo_chomp.launch
```

Vervolgens start ook het Python programma. De planner berekent een pad van start- naar eindconfiguratie, rekening houdend met het obstakel in de omgeving. Dit pad wordt voorgesteld in Figuur 68.



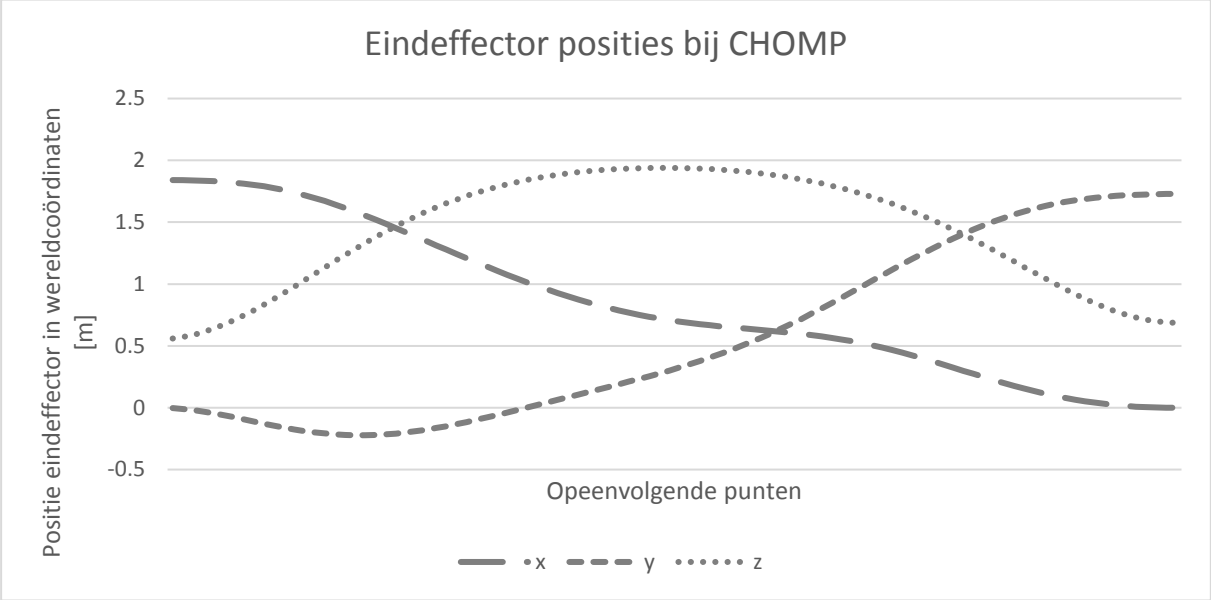
Figuur 68: Het berekende traject bij CHOMP.

In Figuur 68 voert CHOMP een correcte beweging uit. Het obstakel in de omgeving wordt vermeden. Tijdens het plannen worden alle jointposities, alsook jointsnelheden en acceleraties weggeschreven van het gehele traject. In Figuur 69 worden de jointposities weergegeven van de berekende baan uit Figuur 68.



Figuur 69: Jointposities bij CHOMP.

Verder worden hier ook de posities van de eindeffector berekend met voorwaartse kinematica. Deze wordt geplot in Figuur 70, dit is de eindeffectorpositie in xyz-coördinaten.



Figuur 70: Posities van de eindeffector bij CHOMP.

In Figuur 70 worden de posities van de eindeffector weergegeven. Deze komen overeen met het traject uit Figuur 68, waarbij het obstakel in de omgeving wordt vermeden. Met de standaardparameters vindt CHOMP enkel een traject als er geen obstakels in de omgeving zijn. De parameters werden dus aangepast om toch te kunnen plannen met obstakels, deze worden weergegeven in Figuur 71.

```
planning_time_limit: 120.0
max_iterations: 2500
max_iterations_after_collision_free: 20
obstacle_cost_weight: 1.0
learning_rate: 100.0
ridge_factor: 0.0
collision_clearence: 0.1
collision_threshold: 0.07
use_stochastic_descent: true
enable_failure_recovery: false
max_recovery_attempts: 10
trajectory_initialization_method: "cubic"
```

Figuur 71: Aangepaste parameters voor CHOMP.

De parameter “max iterations” in Figuur 71 werd verhoogd in waarde. Zo is het toegelaten dat CHOMP meer berekeningen maakt om een oplossing voor het padplanningsprobleem te vinden. Verder werd ook de parameter “learning rate” verhoogd in waarde. Deze parameter is een waarde voor de hoeveelheid energie dat CHOMP gebruikt om een lokaal minimum te herkennen. De “ridge factor” blijft ingesteld op 0.0. In deze toepassing is het niet nodig om ruis toe te voegen aan het traject, want de obstakels worden vermeden. Deze parameters werden tevens al uitgelegd in 2.3.2.

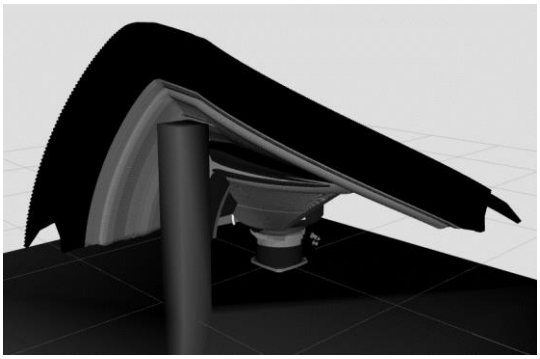
Bij CHOMP is zijn gegenereerde traject, in tegenstelling tot RRTConnect uit OMPL, altijd identiek. De conclusie is dat CHOMP beter werkt als STOMP. Het gegenereerde traject van CHOMP is vloeiender dan die van STOMP. De standaardparameters werden aangepast om met obstakels in de omgeving te kunnen werken. Met de aangepaste parameters wordt het obstakel correct vermeden.

6.4 Point-to-point planning met TrajOpt

Dit deelhoofdstuk evalueert de planner TrajOpt om van configuratie A naar B te bewegen.
Om de simulatie met de KR5-robot op te starten heeft TrajOpt nog een extra package nodig, namelijk "Tesseract". Deze package zorgt ervoor dat TrajOpt bruikbaar is binnen RVIZ. Met volgend commando start de planner.

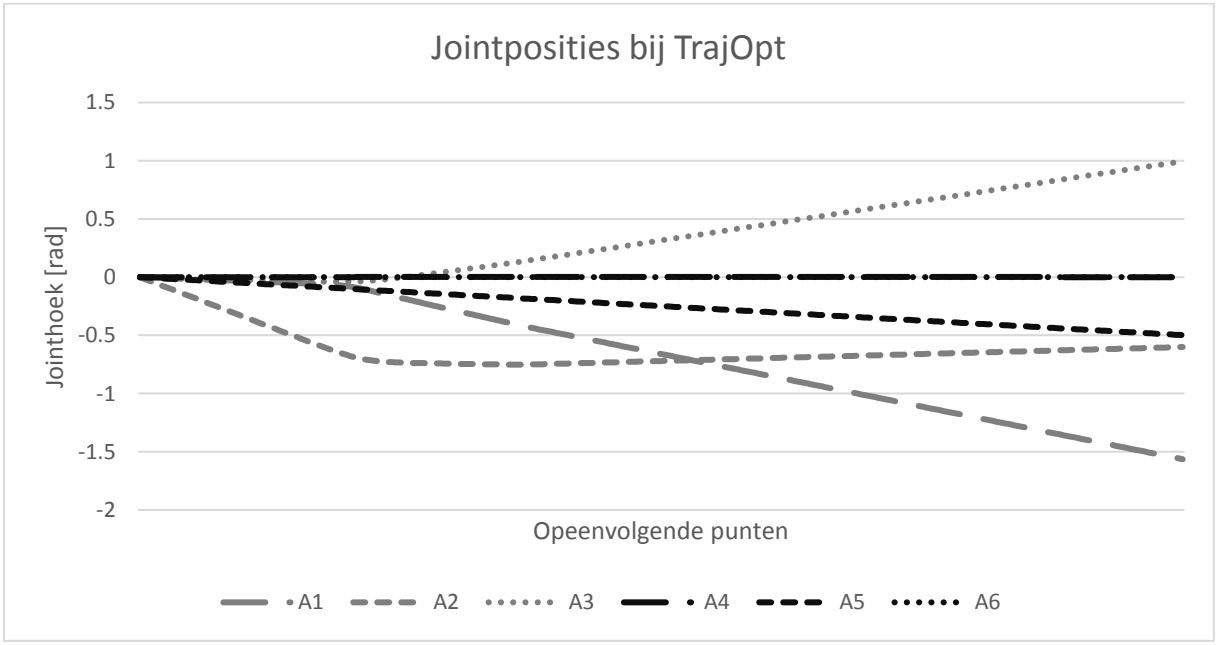
```
roslaunch trajopt_examples load_kr5.launch
```

Vervolgens start hier een cpp-bestand. De planner berekent een pad van start- naar eindconfiguratie, rekening houdend met het obstakel in de omgeving. Dit obstakel werd toegevoegd als STL-bestand aan de scène van Tesseract. Het pad wordt voorgesteld in Figuur 72.



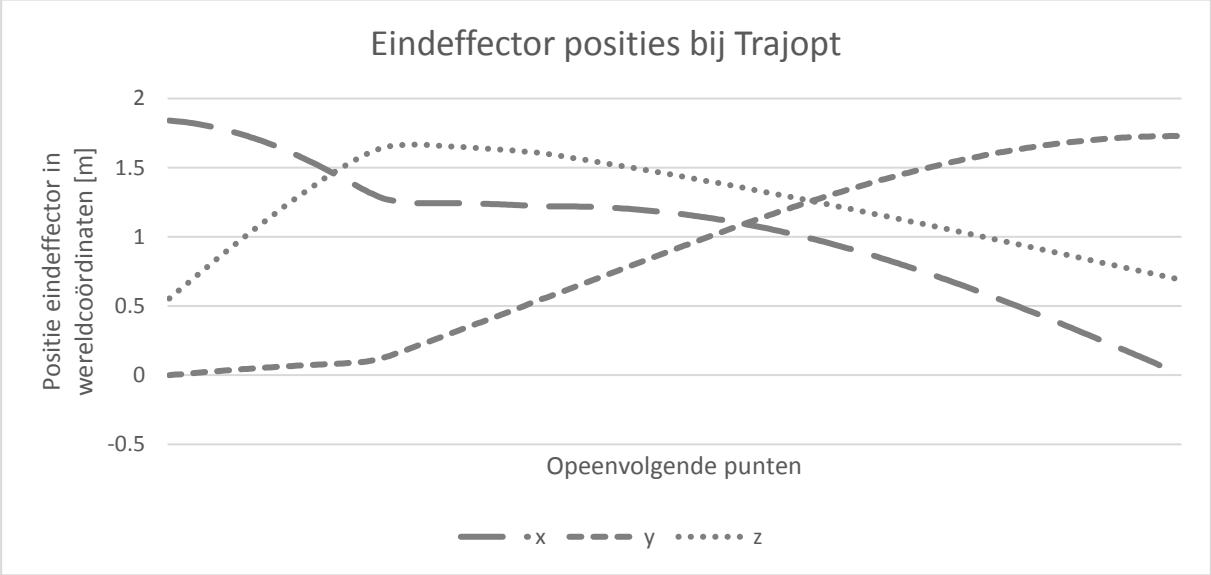
Figuur 72: Het berekende traject bij TrajOpt.

In Figuur 72 voert TrajOpt een correcte beweging uit. Het obstakel in de omgeving wordt vermeden. Tijdens het plannen worden enkel jointposities weggeschreven van het traject. In Figuur 73 worden deze jointposities weergegeven van de berekende baan uit Figuur 72.



Figuur 73: Jointposities bij TrajOpt.

Verder worden hier ook de posities van de eindeffector berekend met voorwaartse kinematica. Deze wordt geplot in Figuur 74, dit is de eindeffectorpositie in xyz-coördinaten.



Figuur 74: Posities van de eindeffector bij TrajOpt.

In Figuur 74 worden de posities van de eindeffector weergegeven. Deze komen overeen met het traject uit Figuur 72, waarbij het obstakel in de omgeving wordt vermeden. De parameters voor TrajOpt werden ingesteld om het obstakel ten alle tijden te vermijden. Deze worden weergegeven in Figuur 75.

```
pci->basic_info.n_steps = 300;
pci->basic_info.manip = "manipulator";
pci->basic_info.start_fixed = true;
pci->basic_info.use_time = false;
pci->opt_info.max_iter = 1000;
pci->opt_info.min_approx_improve = 1e-3;
pci->opt_info.min_trust_box_size = 1e-3;
```

Figuur 75: Ingestelde parameters voor TrajOpt.

De parameter “n_steps” in Figuur 75 werd verhoogd in waarde. Zo is het toegelaten dat TrajOpt meer tussenconfiguraties berekent. Verder werden er ook constraints en kostenfuncties gedefinieerd. Zo wordt er een constraint opgelegd voor het eindconfiguratie, zodat deze zeker bereikt wordt. Als kostenfunctie wordt de obstakeldetectie voor TrajOpt gedefinieerd.

Bij TrajOpt is het gegenereerde traject, in tegenstelling tot RRTConnect uit OMPL, altijd identiek.

De conclusie is dat TrajOpt goed werkt in deze case. Er werd een pad gevonden van begin- tot eindconfiguratie in deze omgeving. De parameters werden nauwelijks aangepast om in deze omgeving correct te kunnen functioneren. Met de constraints en de kostenfuncties kan het pad beïnvloed worden, zodanig dat het voldoet aan de vooropgestelde eisen.

6.5 Conclusie

De verschillende planners werden in dit hoofdstuk onderzocht en getest op de KR5-robot. In dit deelhoofdstuk wordt een overzicht van bevindingen besproken.

RRTConnect van OMPL werkt goed. Er werd altijd een pad gevonden van begin- tot eindconfiguratie in deze omgeving. De standaardparameters werden niet aangepast en het obstakel in de omgeving wordt vermeden. Een nadeel is echter dat er iedere keer een ander traject als oplossing wordt gegenereerd.

STOMP werkt ook goed, als de standaardparameters worden aangepast. De parameters "number of iterations" en "stddev" werden verhoogd in waarde. Deze werden aangepast om met het obstakel in de omgeving te kunnen werken. Zo wordt er een oplossing gevonden en het obstakel wordt vermeden. Bij STOMP kan je per joint de ruis instellen, wat bijvoorbeeld handig kan zijn in een andere toepassing. Zo kan bijvoorbeeld toegelaten/verboden worden om het polsgewricht te bewegen tijdens het plannen. STOMP zijn traject is tevens ook veel vloeiender dan dat van RRTConnect. Deze baan is dan ook meer geschikt om de robot in het echt aan te sturen.

CHOMP werkt ook goed, als de standaardparameters worden aangepast. Deze parameters werden aangepast om met het obstakel in de omgeving te kunnen werken. Zo wordt er een oplossing gevonden en het obstakel wordt vermeden. Deze planner lijkt in deze omgeving veel op STOMP, het traject is eveneens vloeiender dan dat van RRTConnect.

TrajOpt werkt eveneens goed, de parameters werden hier nauwelijks aangepast. Er werd een pad gevonden van begin- tot eindconfiguratie in de omgeving. Met de constraints en de kostenfuncties kan het pad beïnvloed worden, zodanig dat het voldoet aan de vooropgestelde eisen. In deze masterproef werden enkel positie constraints gedefinieerd. De kostenfuncties die gedefinieerd werden zijn voor jointsnelheid, -acceleratie, -jerk en obstakeldetectie.

De conclusie is dat alle planners goed werken om van A naar B te bewegen, rekening houdend met de omgeving van de robot. Voor zeer complexe omgevingen (met veel obstakels) is RRTConnect het meest geschikt (dit kan geconcludeerd worden uit de tutorial van MoveIt!, waarbij vier obstakels in de omgeving zijn). Deze standaardparameters worden niet aangepast om te werken met obstakels in de omgeving. Als de omgeving minder complex en gekend is, kan met STOMP of CHOMP een vloeiender pad gegenereerd worden, mits de parameters worden aangepast. Met de TrajOpt planner kunnen er constraints en kostenfuncties toegevoegd worden. Hiermee houdt TrajOpt dan rekening tijdens het genereren van een pad. Constraints of kostenfuncties zijn bij de andere planners niet voorzien, wat TrajOpt een voordeel geeft qua aansturing van de robot.

7 Evaluatie van de padplanners voor point-to-point bewegingen met de UR5-robot

In dit hoofdstuk worden de bekomen trajecten vergeleken en getest op een echte UR5-robot. Tevens wordt ook een geprogrammeerde zigzagbeweging via een Python programma getest. De bewegingen vanuit de simulatiesoftware worden direct gelinkt naar bewegingen op de echte robot. Om dit te realiseren moet de Universal Robot package geïnstalleerd worden met volgende commando's [24].

```
sudo apt-get install ros-kinetic-universal-robot

cd ~/ws_moveit/src
git clone -b kinetic-devel https://GitHub.com/ros-industrial/universal_robot.git

cd ~/ws_moveit
rosdep update
rosdep install --from-paths src --ignore-src --rosdistro kinetic

catkin build
source ~/ws_moveit/devel/setup.bash
```

Van de geïnstalleerde package hebben we enkel de instellingen voor de UR5-robot nodig. Eerst moet het IP-adres van de robot juist worden ingesteld. Deze instellingen gebeuren op de Touch-panel van de robot.

```
IP-address robot: 192.168.0.1
Subnet mask: 255.255.255.0
Default gateway: 0.0.0.0
```

Als bovenstaande parameters ingesteld zijn kan er op een eenvoudige manier gecontroleerd worden, met onderstaand commando, of er een verbinding is tussen de robot en de computer.

```
ping 192.168.0.1
```

Vervolgens moet er een verbinding opgesteld worden tussen de robot en ROS. Deze connectie wordt opgestart door volgend commando.

```
roslaunch ur_bringup ur5_bringup.launch robot_ip:=192.168.0.1
```

Daarna wordt het volgende commando uitgevoerd om motion planning toe te laten.

```
roslaunch ur5_moveit_config ur5_moveit_planning_execution.launch
```

Als voorlaatste stap kan de simulatie opgestart worden. Deze is dan gelinkt aan de echte robot.

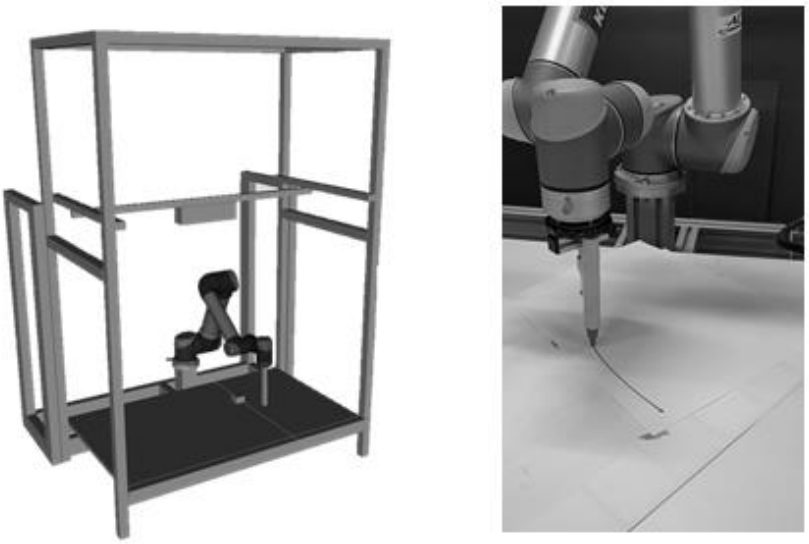
```
roslaunch ur5_moveit_config moveit_rviz.launch config:=true
```

Als laatste stap kan dan het Python programma opgestart worden.

```
rosrun ur5_moveit_config zigzag_robot.py
```

Dit programma stuurt de bewegingen aan in de simulatie. Doordat de simulatie nu gekoppeld is aan de echte robot, wordt het traject ook effectief uitgevoerd op de UR5-robot [30].

Er werd ook een obstakel toegevoegd aan de omgeving van de robot, weergegeven in Figuur 76. Dit obstakel is een mesh die nagemaakt werd t.o.v. de echte opstelling. Zo kan tijdens het plannen van trajecten rekening gehouden worden met deze omgeving.



Figuur 76: UR5-robot en simulatieomgeving.

De posities van de eindeffector worden in deze case met elkaar vergeleken. De snelheid wordt niet onderzocht in deze toepassing.

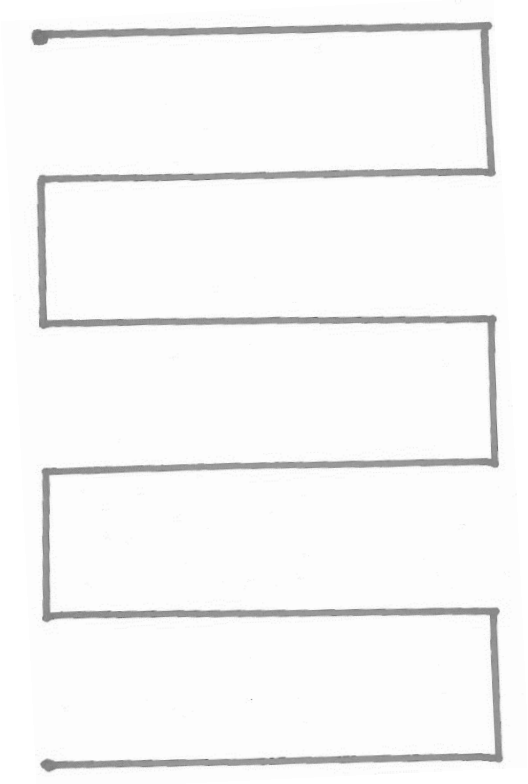
7.1 Point-to-point planning met Cartesische interpolatie

Deze methode gebruikt waypoints of tussenpunten om het pad te realiseren. Met behulp van een Python programma wordt de eindeffector van de UR5-robot in xyz-coördinaten naar zijn einddoel gestuurd. De Cartesische interpolatie zorgt ervoor dat binnenin het opgegeven begin- en eindconfiguratie lineaire interpolatie wordt toegepast om tussenpunten te creëren. Deze tussenpunten zijn ingesteld op een lineaire afstand onderling van 1 millimeter.

De link tussen de simulatie en de echte robot werd al opgestart. Aan de robot werd een stift gemonteerd om het gevolgde traject visueel voor te stellen. Door een Python bestand uit te voeren, met het volgende commando, wordt de zigzagbeweging uitgevoerd op de UR5-robot.

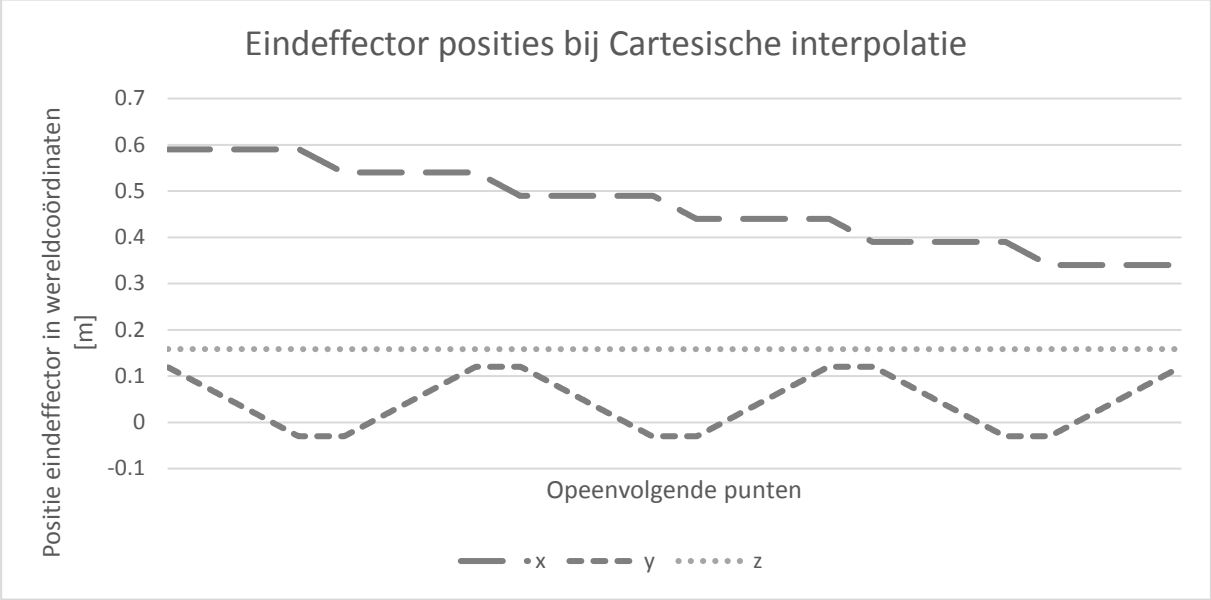
```
rosrun ur5_moveit_config zigzag_robot.py
```

Het vorige commando voert het Python bestand uit dat ervoor zorgt dat de planner begint met de zigzagbeweging. Het traject werd visueel getekend met een stift als eindeffector op een A4-blad, weergegeven in Figuur 77.



Figuur 77: Zigzagbeweging op de UR5-robot met Cartesische interpolatie.

De positie van de eindeffector wordt ook hier berekend met voorwaartse kinematica. De zigzagbeweging wordt ook geplot in Figuur 78, identiek aan de beweging uit Figuur 77.



Figuur 78: Posities van de eindeffector bij Cartesische interpolatie.

In Figuur 78 voert Cartesische interpolatie een perfecte zigzagbeweging uit. Omdat de zigzagbeweging relatief eenvoudig is, kan de Cartesische interpolatie hier goed mee overweg.

De conclusie is idem als in hoofdstuk 5.6. Cartesische interpolatie werkt zeer goed tot nagenoeg perfect voor het berekenen van de posities van de eindeffector tijdens de zigzagbeweging. Tussen de begin- en eindconfiguraties worden telkens waypoints berekend met een opgegeven resolutie. Ook de oriëntatie van de spuitkop blijft altijd constant bij de waypoints. Er is geen obstakeldetectie ingebouwd in deze planner. Het uitgevoerde traject op de echte robot is identiek als het traject in de simulatie.

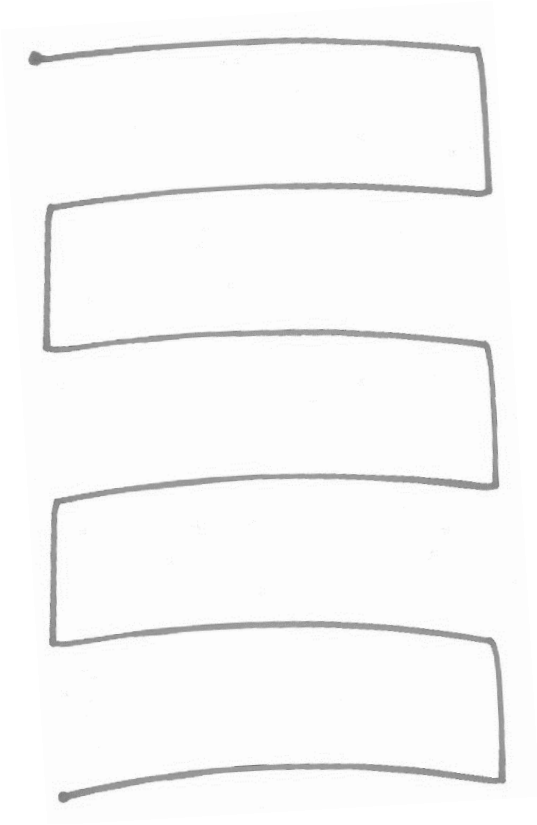
7.2 Point-to-point planning met RRTConnect

De RRTConnect planner gebruikt de bibliotheek OMPL om van configuratie A naar B te bewegen. Met behulp van een Python programma wordt de eindeffector van de UR5-robot in xyz- en rpy-coördinaten naar zijn einddoel gestuurd. Bij de Cartesische interpolatie werd gebruik gemaakt van tussenpunten, maar dat is hier niet het geval.

De link tussen de simulatie en de echte robot werd al eerder opgestart. Door een Python bestand uit te voeren, met het volgende commando, wordt de zigzagbeweging uitgevoerd op de UR5-robot.

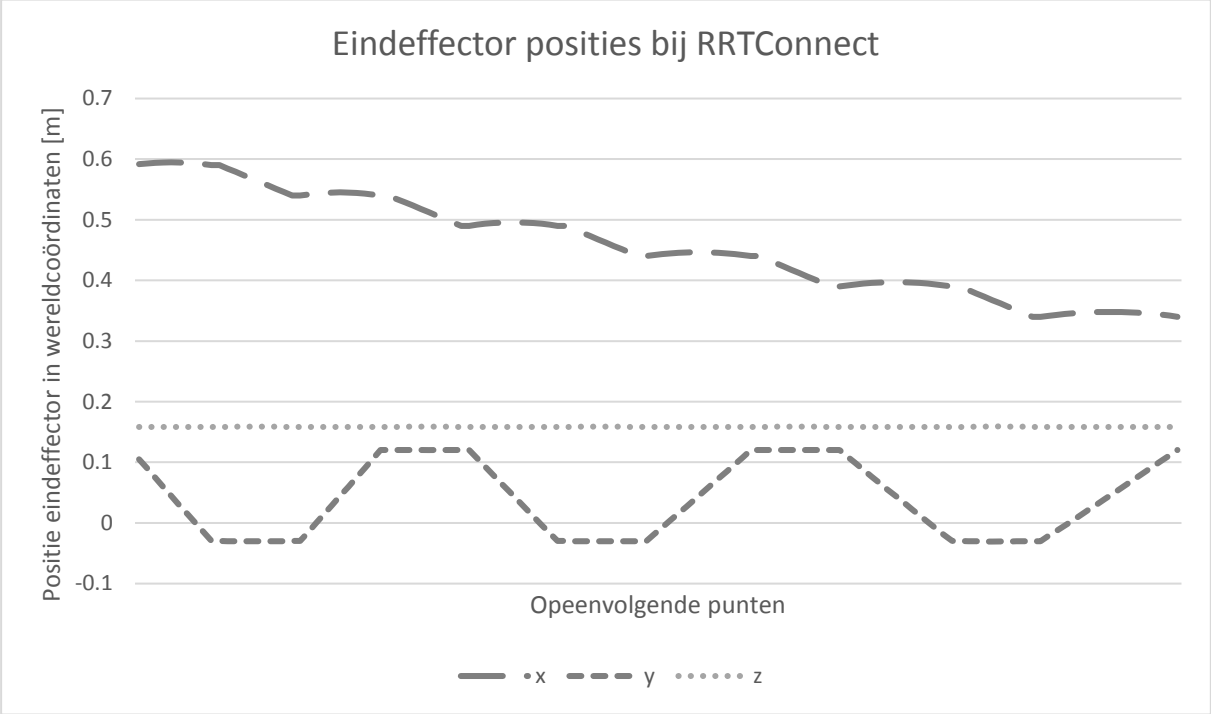
```
rosrun ur5_moveit_config zigzag_robot.py
```

Het vorige commando voert het Python bestand uit dat ervoor zorgt dat de planner begint met de zigzagbeweging. Het traject werd visueel getekend met een stift als eindeffector op een A4-blad, weergegeven in Figuur 79.



Figuur 79: Zigzagbeweging op de UR5-robot met RRTConnect.

De positie van de eindeffector wordt ook hier berekend met voorwaartse kinematica. De zigzagbeweging wordt ook geplot in Figuur 80, identiek aan de beweging uit Figuur 79.

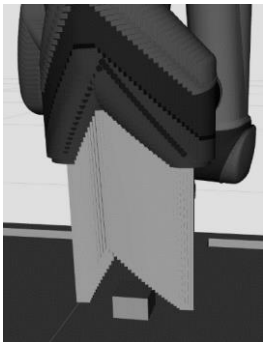


Figuur 80: Posities van de eindeffector bij RRTConnect.

In Figuur 80 voert de planner een minder goede zigzagbeweging uit. Het traject is niet rechtlijnig genoeg. De planner berekent wel altijd een botsingsvrij traject.

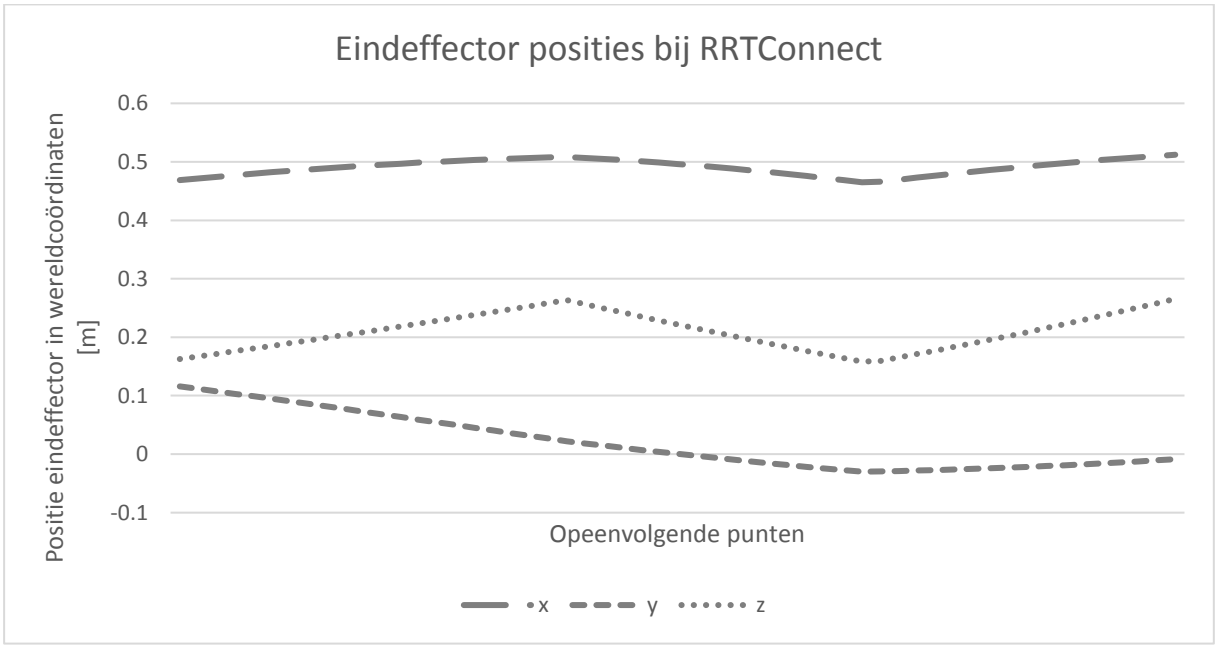
De conclusie is dat RRTConnect uit OMPL goed werkt in deze case. Er wordt altijd een pad gevonden van begin- tot eindconfiguratie in deze omgeving. De standaardparameters in de package worden dan ook niet aangepast. De posities worden goed nagestreefd, alleen is het traject niet rechtlijnig. Bij de Cartesische interpolatie methode was het traject wel rechtlijnig. Het uitgevoerde traject op de echte robot is identiek als het traject in de simulatie.

In deze case wordt ook besproken hoe de planner omgaat met obstakels in zijn omgeving. Om dat te testen werd een Python programma geschreven dat slechts één deel uitvoert van de zigzagbeweging. Er werd een obstakel toegevoegd dat de robot moet vermijden. In Figuur 81 wordt van configuratie A (rechts) naar B (links) bewogen.



Figuur 81: RRTConnect met een obstakel in de omgeving.

RRTConnect gaat goed over het obstakel, weergegeven in Figuur 81. In het traject bevinden zich wel knikken die voor grote versnellingen kunnen zorgen. Dit is eigen aan een sample gebaseerde planner. De knikken worden ook weergegeven in de z-coördinaten in Figuur 82.



Figuur 82: Posities van de eindeffector bij RRTConnect.

Er werden verschillende trajecten gesimuleerd en elke oplossing is anders. Vaak gaat de pen veel hoger dan nodig om het obstakel te vermijden en zit er vaak een knik in het traject. Het gegenereerde traject is echter wel aanvaardbaar, want het obstakel wordt altijd vermeden. Maar het resultaat is niet optimaal.

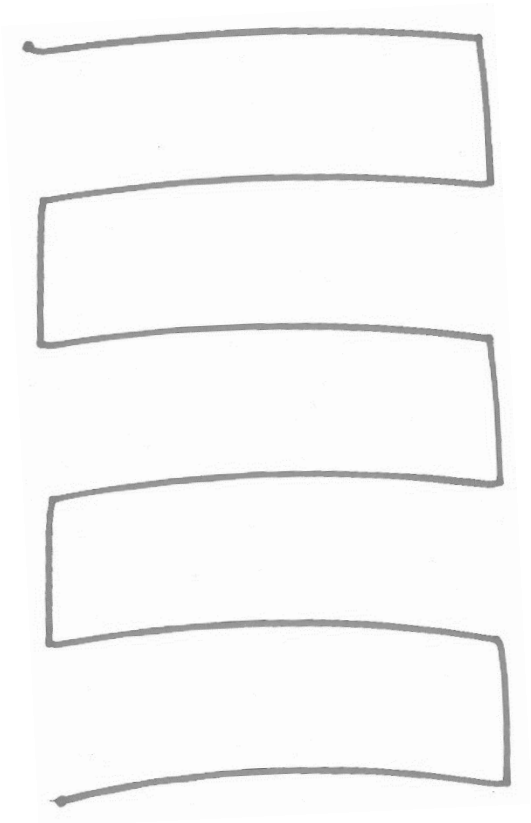
7.3 Point-to-point planning met STOMP

STOMP is een optimalisatie gebaseerde padplanner die gebruikt wordt om van configuratie A naar B te bewegen. Met behulp van een Python programma wordt de eindeffector van de UR5-robot in xyz- en rpy-coördinaten naar zijn einddoel gestuurd. Bij de Cartesische interpolatie werd gebruik gemaakt van tussenpunten, maar dat is hier niet het geval.

De link tussen de simulatie en de echte robot werd al eerder opgestart. Door een Python bestand uit te voeren, met het volgende commando, wordt de zigzagbeweging uitgevoerd op de UR5-robot.

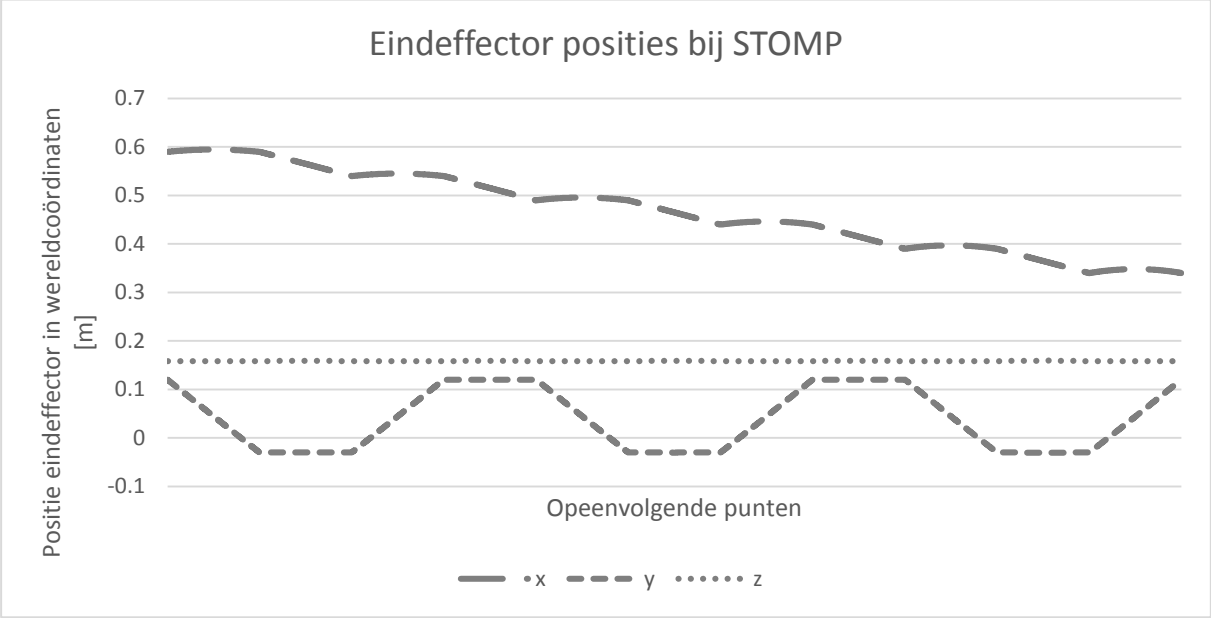
```
rosrun ur5_moveit_config zigzag_robot.py
```

Het vorige commando voert het Python bestand uit dat ervoor zorgt dat de planner begint met de zigzagbeweging. Het traject werd visueel getekend met een stift als eindeffector op een A4-blad, weergegeven in Figuur 83.



Figuur 83: Zigzagbeweging op de UR5-robot met STOMP.

De positie van de eindeffector wordt ook hier berekend met voorwaartse kinematica. De zigzagbeweging wordt ook geplot in Figuur 84, identiek aan de beweging uit Figuur 83.

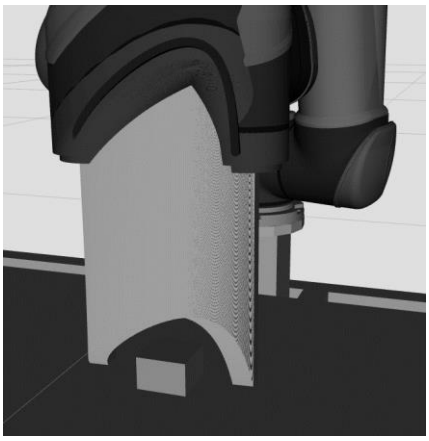


Figuur 84: Posities van de eindeffector bij STOMP.

In Figuur 84 voert STOMP een minder goede zigzagbeweging uit dan Cartesische interpolatie. Het traject is niet rechtlijnig genoeg. De planner berekent wel altijd een botsingsvrij traject.

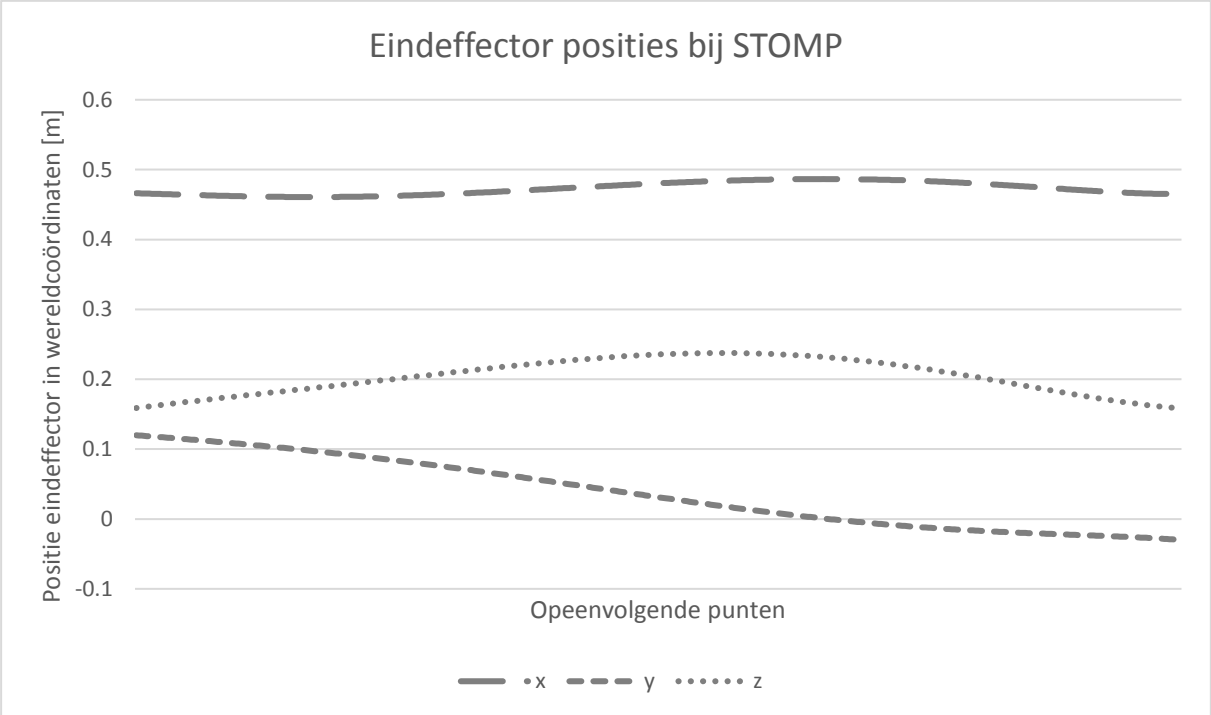
De conclusie is dat STOMP goed werkt in deze case. Er wordt altijd een pad gevonden van begin- tot eindconfiguratie in deze omgeving. De standaardparameters in de package worden dan ook niet aangepast. Dit komt omdat de standaardparameters van STOMP in de UR5-package juist zijn ingesteld op GitHub. In tegenstelling tot de verfrobot en de KR5-robot waar deze parameters manueel werden ingesteld. De posities worden goed nagestreefd, alleen is het traject niet rechtlijnig genoeg. Het uitgevoerde traject op de echte robot is identiek als het traject in de simulatie.

In deze case wordt ook besproken hoe de planner omgaat met obstakels in zijn omgeving. Om dat te testen werd een Python programma geschreven dat slechts één deel uitvoert van de zigzagbeweging. Er werd een obstakel toegevoegd dat de robot moet vermijden. In Figuur 85 wordt van configuratie A (rechts) naar B (links) bewogen.



Figuur 85: STOMP met een obstakel in de omgeving.

STOMP gaat perfect over het obstakel, weergegeven in Figuur 85. In dit traject bevindt zich geen knik, want de gegenereerde trajecten van deze planner zijn veel vloeiender dan die van RRTConnect. Het traject wordt ook weergegeven in Figuur 86.



Figuur 86: Posities van de eindeffector bij STOMP.

Er werden verschillende trajecten gesimuleerd en elke oplossing is bijna hetzelfde. Vaak beweegt de stift kort en vloeiend over het object. Het gegenereerde traject, om het obstakel te vermijden, is hier wel aanvaardbaar.

7.4 Point-to-point planning met CHOMP

De CHOMP planner is ook een optimalisatie gebaseerde planner. De zigzagbeweging kan niet worden uitgevoerd omdat CHOMP niet kan werken met pose goals. De terminal geeft aan dat de planner enkel met jointgoals kan werken. De conclusie is idem als in hoofdstuk 5.4. CHOMP kan niet werken met xyz-coördinaten.

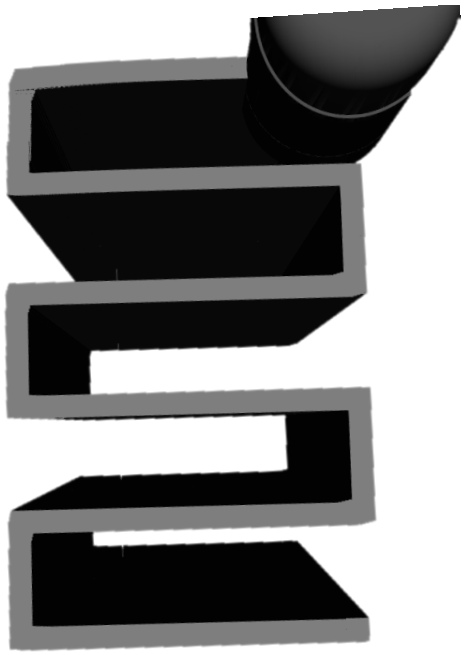
7.5 Point-to-point planning met TrajOpt

TrajOpt is ook een optimalisatie gebaseerde padplanner die gebruikt wordt om van configuratie A naar B te bewegen. Met behulp van een cpp-programma wordt de eindeffector van de UR5-robot in xyz- en rpy-coördinaten naar zijn einddoel gestuurd.

TrajOpt werd enkel in simulatie getest. Omdat er enkel posities worden gegenereerd door TrajOpt is het onmogelijk om deze zonder snelheden en acceleraties door te sturen naar de robot. Om de robot toch op te starten, in simulatie, is volgend commando nodig.

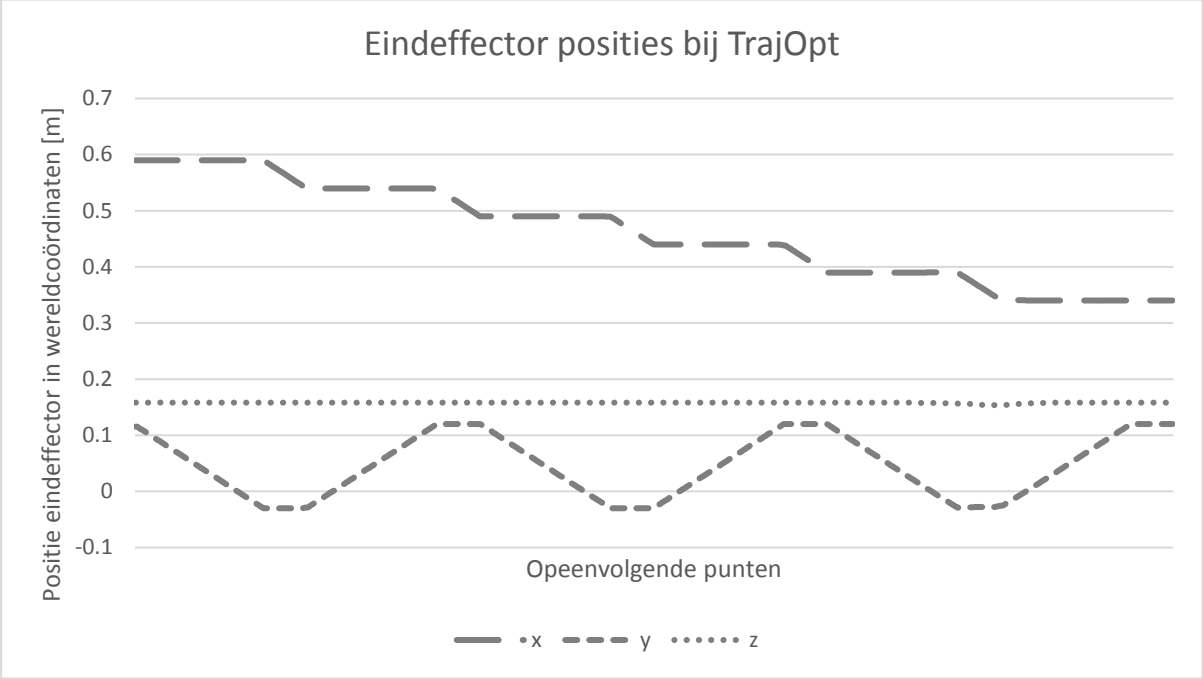
```
roslaunch trajopt_examples load_ur5_zigzag.launch
```

Met deze launch-file wordt vervolgens ook een cpp-programma uitgevoerd, die ervoor zorgt dat de zigzagbeweging door TrajOpt wordt berekend. Het berekende traject werd visueel met markers voorgesteld in de simulatie. Deze weergave wordt voorgesteld in 3D om de diepte van de stift voor te stellen. Dit is weergegeven in Figuur 87.



Figuur 87: Zigzagbeweging op de UR5-robot met TrajOpt.

De positie van de eindeffector wordt ook hier berekend met voorwaartse kinematica. De zigzagbeweging wordt ook geplot in Figuur 88, identiek aan de beweging uit Figuur 87.



Figuur 88: Posities van de eindeffector bij TrajOpt.

In Figuur 88 voert de planner een perfecte zigzagbeweging uit. Het gegenereerde traject is rechtlijnig. De planner berekent echter geen botsingsvrij pad, omdat TrajOpt gebruik maakt van constraints. Elk tussenpunt, dat als constraint gedefinieerd wordt, wordt door de robot uitgevoerd. Als zo'n tussenpunt in een obstakel ligt, zal TrajOpt geen alternatief traject berekenen.
TrajOpt geeft wel altijd aan dat het traject door een obstakel gaat.

De conclusie is dat TrajOpt uitstekend werkt voor het berekenen van de posities. Er wordt altijd een pad gevonden van begin- tot eindconfiguratie in deze omgeving. De parameters voor TrajOpt worden niet aangepast. Obstakelvrij plannen gaat niet met constraints, als de tussenpunten te kort bij elkaar liggen. Deze planner geeft wel aan dat het traject niet obstakelvrij is.

7.6 Conclusie

De verschillende planners werden onderzocht en getest op de UR5-robot in dit hoofdstuk. In dit deelhoofdstuk wordt een overzicht van bevindingen besproken.

Cartesische interpolatie werkt uitstekend voor het berekenen van de posities tijdens de zigzagbeweging. Deze methode maakt gebruik van waypoints die worden berekend met een opgegeven resolutie. Er is echter geen obstakeldetectie ingebouwd in deze planner.

RRTConnect uit OMPL werkt voor deze robot goed met standaardparameters. De posities van de zigzagbeweging worden goed nagestreefd, maar het traject is niet rechtlijnig. Deze planner genereert wel obstakelvrije trajecten. De pen, die gemonteerd werd als eindeffector, gaat echter vaak veel hoger dan nodig om het obstakel te vermijden. Vaak zit er ook een knik in het traject, wat hoge accelerations met zich meebrengt. Dit kan de levensduur van de robot beïnvloeden. De gegenereerde trajecten zijn tevens bij elke simulatie-run anders, wat niet wenselijk is in de industrie. Het gegenereerde traject is aanvaardbaar maar niet optimaal.

STOMP werkt goed met de standaardparameters uit de package van de UR5-robot. De posities van de zigzagbeweging worden goed nagestreefd, maar ook hier is het traject niet rechtlijnig. De planner genereert wel obstakelvrije trajecten. De pen gaat, nagenoeg perfect, in een boogje over het obstakel. Het gegenereerde traject is veel vloeiender dan het pad van RRTConnect. Het gegenereerde traject en het resultaat is hier wel aanvaardbaar.

CHOMP kan niet werken met xyz-coördinaten, enkel met jointgoals. CHOMP werd niet verder onderzocht voor de UR5-robot.

TrajOpt berekent een perfecte zigzagbeweging. Het traject is rechtlijnig, maar de planner berekent geen botsingsvrij pad. De parameters voor deze planner werden niet aangepast. TrajOpt maakt gebruik van constraints voor de tussenposities, waardoor de robot al deze tussenpunten wil aanvaren. TrajOpt geeft wel aan dat het gegenereerde traject niet obstakelvrij is.

De conclusie van deze case is dat STOMP het meest geschikt is voor de UR5-robot (voor point-to-point bewegingen), ondanks dat dit traject niet rechtlijnig is.

8 Besluit

In deze masterproef werden drie cases opgesteld (verfrobot, KR5-robot, UR5-robot) en vijf padplanners werden hierop geëvalueerd. De geëvalueerde padplanners die besproken werden zijn: RRTConnect uit de bibliotheek van OMPL, STOMP, CHOMP, Cartesische interpolatie en TrajOpt.

Elke planner heeft zijn voor- en nadelen die per case en in verschillende omgevingen werden besproken in deze masterproef. De hoofdstukken met de conclusies (5.6, 6.5 en 7.6) geven een beknopt overzicht met de voor- en nadelen van elke planner. Algemeen kan er geconcludeerd worden dat de padplanners slagen in hun opzet: het genereren van een botsingsvrij pad. Sommige trajecten bevatten knikken/schokken wat het eindresultaat niet optimaal maakt. Ze kunnen ook vaak geen lineair traject genereren waardoor het eindresultaat niet optimaal is.

Een mogelijke oplossing voor dit probleem is om twee padplanners te combineren met elkaar. Zo is het bijvoorbeeld mogelijk om eerst een pad te laten berekenen door een planner uit de bibliotheek van OMPL, om vervolgens een nabewerking te doen met een optimalisatie gebaseerde padplanner. Op deze manier kan men de nadelen van de ene planner compenseren met voordelen uit een andere planner. Twee padplanners combineren werd in deze masterproef niet getest. Dit is zeker nog een interessant onderwerp voor verder onderzoek.

Referentielijst

[1] KULeuven, "Onderzoekcentrum ACRO," *kuleuven.be*, 2018. [Online]. Available: https://iiw.kuleuven.be/onderzoek/acro. [Accessed: 18-Sep-2018].

[2] J. De Maeyer, B. Moyaers and E. Demeester, "Cartesian Path Planning for Arc Welding Robots : Evaluation of the Descartes Algorithm," *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8, 2017.

[3] ROS, "STOMP Planner," *docs.ros.org*, 2018. [Online]. Available: http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/stomp_planner/stomp_planner_tutorial.html. [Accessed: 30-Nov-2018].

[4] ROS, "CHOMP Planner," *docs.ros.org*, 2018. [Online]. Available: http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/chomp_planner/chomp_planner_tutorial.html. [Accessed: 30-Nov-2018].

[5] ROS, "TrajOpt Planner," *ROS-Industrial*, 2018. [Online]. Available: https://rosindustrial.org/news/2018/7/5/optimization-motion-planning-with-tesseract-and-trajopt-for-industrial-applications. [Accessed: 30-Nov-2018].

[6] ROS, "OMPL Planner," *docs.ros.org*, 2018. [Online]. Available: http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/ompl_interface/ompl_interface_tutorial.html. [Accessed: 30-Nov-2018].

[7] Octomap, "Octomap github," *octomap.github.io*, 2019. [Online]. Available: https://octomap.github.io/. [Accessed: 30-Nov-2018].

[8] K. M. Lynch and F. C. Park, *Modern Robotics - mechanics, planning, and control*. Cambridge University Press, 2017.

[9] A. Gasparetto, P. Boscariol, A. Lanzutti and R. Vidoni, "Path planning and Trajectory Planning Algorithms: A General Overview," *Motion and Operation Planning of Robotic Systems Mechanisms and Machine Science*, pp. 3–27, 2015.

[10] M. Strandberg, "Augmenting RRT-planners with local trees," *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA 04. 2004*, 2004.

[11] H. Choset, "Ch. 07: Sample-Based Motion Planning BT - Robot Motion Planning," *Robot Motion Planning*, pp. 1–109, 2006.

[12] J. L. Blanco, M. Bellone and A. Gimenez-Fernandez, "TP-space RRT - Kinematic path planning of non-holonomic any-shape vehicles," *International Journal of Advanced Robotic Systems*, vol. 12, no. 5, p. 55, 2015.

[13] J. Schulman, J. Ho, A. Lee, I. Awwal, H. Bradlow and P. Abbeel, "Finding Locally Optimal, Collision-Free Trajectories with Sequential Convex Optimization," *Robotics: Science and Systems IX*, 2013.

[14] B. Kovács, G. Szayer, F. Tajti, M. Burdelis and P. Korondi, "A novel potential field method for path planning of mobile robots by adapting animal motion attributes," *Robotics and Autonomous Systems*, vol. 82, pp. 24–34, 2016.

[15] T. I. P. Enst, M. Anab and S. Ceng, "OPTIMAL TRAJECTORY PLANNING OF MANIPULATORS: A REVIEW," *Journal of Engineering Science and Technology*, vol. 2, no. 1, pp. 32–54, 2009.

[16] Z. Wu, W. Fu, R. Xue and W. Wang, "A novel global path planning method for mobile robots based on Teaching-Learning-Based Optimization," *Information (Switzerland)*, vol. 7, no. 3, 2016.

[17] Karvaki Lab, "Open Motion Planning Library: A Primer," 2015.

[18] OMPL, "OMPL Available Planners," *Physical and Biological Computing Group Department of Computer Science Rice University*, 2018. [Online]. Available: http://ompl.kavrakilab.org/planners.html. [Accessed: 30-Nov-2018].

[19] M. Zucker, N. Ratliff, A. D. Dragan, M. Pivtoraiko, M. Klingensmith, C. M. Dellin, J. A.

Bagnell and S. S. Srinivasa, “CHOMP: Covariant Hamiltonian optimization for motion planning,” *The International Journal of Robotics Research*, vol. 32, no. 9–10, pp. 1164–1193, 2013.

[20] M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor and S. Schaal, “STOMP : Stochastic Trajectory Optimization for Motion Planning,” *2011 IEEE International Conference on Robotics and Automation*, 2011.

[21] J. Schulman, Y. Duan, J. Ho, A. X. Lee, I. Awwal, H. Bradlow, J. Pan, S. Patil, K. Y. Goldberg and P. Abbeel, “Motion planning with sequential convex optimization and convex collision checking,” *The International Journal of Robotics Research*, vol. 33, no. 9, pp. 1251–1270, 2014.

[22] E. Tromme, A. Held, P. Duysinx and O. Brüls, “System-Based Approaches for Structural Optimization of Flexible Mechanisms,” *Archives of Computational Methods in Engineering*, vol. 25, no. 3, pp. 817–844, 2018.

[23] Github, “Kuka experimental github repository,” *Github.com*, 2019. [Online]. Available: https://github.com/ros-industrial/kuka_experimental. [Accessed: 01-Dec-2018].

[24] ROS, “Universal robot,” *wiki.ros.org*, 2019. [Online]. Available: http://wiki.ros.org/universal_robot. [Accessed: 01-Mar-2019].

[25] Github, “Libfreenect2 github repository,” *Github.com*, 2019. [Online]. Available: https://github.com/OpenKinect/libfreenect2. [Accessed: 15-Feb-2019].

[26] Github, “iai_kinect2 github repository,” *Github.com*, 2019. [Online]. Available: https://github.com/code-iai/iai_kinect2. [Accessed: 20-Feb-2019].

[27] D. Meagher, “Geometric Modeling Using Octree Encoding,” *Computer Graphics and Image Processing*, vol. 19, no. 1, pp. 129–147, 1982.

[28] Github, “Octomap github repository,” *Github.com*, 2019. [Online]. Available: https://github.com/OctoMap/octomap.git. [Accessed: 25-Feb-2019].

[29] Devforum, “Roll, Pitch and Yaw,” *devforum.roblox.com*, 2019. [Online]. Available: https://devforum.roblox.com/t/take-out-pitch-from-rotation-matrix-while-preserving-yaw-and-roll/95204. [Accessed: 01-Jun-2019].

[30] ROS, “Integration of Universal robot with ROS,” *wiki.ros.org*, 2019. [Online]. Available: http://wiki.ros.org/universal_robot/Tutorials/Getting Started with a Universal Robot and ROS-Industrial. [Accessed: 05-Mar-2019].