2019•2020

Faculteit Industriële ingenieurswetenschappen

master in de industriële wetenschappen: energie

Masterthesis

Optimalisatie van het industrieel robotfreesproces door filteren en toevoegen van punten in het robotpad

PROMOTOR :

Prof. dr. ir. Johan BAETEN

PROMOTOR :

dr. ir. Wim PERSOONS

Ward Habraken, Jonas Nelis

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: energie, afstudeerrichting automatisering

Gezamenlijke opleiding UHasselt en KU Leuven

UHASSELT KU LEUVEN

UHASSELT KU LEUVEN

2019 • 2020

Faculteit Industriële ingenieurswetenschappen

master in de industriële wetenschappen: energie

Masterthesis

Optimalisatie van het industrieel robotfreesproces door filteren en toevoegen van punten in het robotpad

PROMOTOR :

Prof. dr. ir. Johan BAETEN

PROMOTOR :

dr. ir. Wim PERSOONS

Ward Habraken, Jonas Nelis

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: energie, afstudeerrichting automatisering

UHASSELT

KU LEUVEN

Deze masterproef werd geschreven tijdens de COVID-19 crisis in 2020. Deze wereldwijde gezondheidscrisis heeft mogelijk een impact gehad op de opdracht, de onderzoekshandelingen en de onderzoeksresultaten.

Woord vooraf

Deze masterproef werd uitgevoerd door Jonas Nelis en Ward Habraken in 2019-2020, als afsluiting van de master in industriële ingenieurswetenschappen energie-automatisering. Deze opleiding is een gezamenlijke opleiding aan de Universiteit Hasselt in samenwerking met de Katholieke Universiteit Leuven. De masterproef is uitgevoerd in opdracht van KUKA Automatisering + Robots N.V te Houthalen.

Graag willen wij van dit voorwoord gebruik maken om enkele personen te bedanken die ons geholpen en begeleid hebben bij het verwezenlijken van onze masterproef. Allereerst willen wij onze interne promotor prof. dr. ir. Johan Baeten van de KU Leuven en externe promotor dr. ir. Wim Persoons van KUKA Automatisering + Robots N.V bedanken voor hun begeleiding en expertise.

Ten tweede bedanken wij KUKA Automatisering + Robots N.V voor het ter beschikking stellen van de nodige componenten zoals de freesrobot en de nodige opleiding voor het programmeren van industriële KUKA robots. Hierbij willen wij Jan Goetschalckx bedanken voor het geven van de lessen tijdens deze opleiding.

Ten derde willen wij Koen Jordens, Carlo Schalley en de andere werknemers binnen KUKA bedanken omdat zij op elk moment openstonden om onze vragen te beantwoorden. Tenslotte bedanken we graag alle docenten die ons doorheen de opleiding de verschillende disciplines van de industriële ingenieurswetenschappen hebben aangeleerd alsook het aanpakken van projecten en problemen.

Inhoudsopgave

Lijst van tabellen

Lijst van figuren

Verklarende woordenlijst

Afkortingen

2D	Tweedimensionaal
3D	Driedimensionaal
BIN	Binair bestandstype
C#	Programmeertaal
CAD	Computer-aided design
CAM	Computer-aided manufacturing
CNC-machine	Computer numerical control
CPU	Central processing unit
CSV	Comma-separated values bestandstype
DPT	Douglas-Peucker tolerantie
Eindeffector	Werktuig aan het uiteinde van de robotarm
GUI	Graphical user interface
O(n)-algoritme	Een algoritme met een lineaire complexiteit, de tijd stijgt lineair met het aantal input punten
PTP	Point to point
TCP	Tool center point
TXT	Tekst bestandstype
XML	Extensible markup language bestandstype

Letters

a	Versnelling	m/s^2
A	Oppervlakte	mm^2
d	Afstand	mm
di	Afstand tussen 2 punten	mm
h	Hoogte	mm
t	Tijd	s
v	Snelheid	m/s

Abstract

KUKA is een producent van industriële robots die ook nieuwe technologieën ontwikkelt voor deze robots. Deze masterproef heeft als doel het versnellen van een industrieel robotfreesproces waarbij de nauwkeurigheid binnen een gegeven tolerantie blijft. De industriële robot moet een pad van punten volgen gebruikmakend van zowel lineaire bewegingen met luswerking als *spline*-bewegingen.

Om het freesproces te versnellen, moeten de punten gefilterd worden. Om de nauwkeurigheid binnen de gegeven tolerantie te houden, moeten ook punten toegevoegd worden waar nodig. Hierna moet de fout van het nieuwe pad berekend worden ten opzichte van het originele pad. Als laatste moet ook de tijdsverbetering van het freesproces berekend kunnen worden.

Eerst wordt een programma geschreven in C# voor het filteren van punten indien de robot gebruik maakt van een lineaire beweging met luswerking. Hiervoor wordt het Douglas-Peucker-filteralgoritme geïmplementeerd. Ook is het belangrijk dat het programma rekening houdt met robotfactoren zoals de luswerking, de oriëntatie van de eindeffector en de spindel. Na het testen van het programma in 2D en 3D, wordt het programma verder geoptimaliseerd zodat het ook punten kan filteren indien de robot gebruik maakt van spline-bewegingen. De tijdsverbetering is afhankelijk van het aantal originele punten en de tolerantie. Testen wezen uit dat snelheidsverbeteringen tot 74% mogelijk zijn indien er een tolerantie van 0,1 mm gebruikt wordt.

Abstract in English

KUKA is a manufacturer of industrial robots and develops new technologies that make use of these industrial robots. The aim of this master's thesis is to accelerate an industrial robot milling process in which the accuracy remains within a given tolerance. The industrial robot has to follow a path of points using both a linear movement with approximate positioning and a spline movement.

In order to accelerate the milling process, the points must be filtered. To keep the accuracy within the given tolerance, points must also be added where necessary. Hereafter the error of the new path must be calculated relative to the original path. Lastly the time gain of the milling process must be calculated.

First, a program is written in C# for filtering points in case the robot uses a linear motion with approximate positioning. Therefore the Douglas-Peucker filter algorithm is implemented. It is also important that the program takes into account robot factors such as approximate positioning, the orientation of the end effector and the spindle state. After testing the program in 2D and 3D, the program will be further optimized so that it can also filter points in case the robot uses a spline motion. The time gain depends on the number of original points and the tolerance. Tests showed that speed improvements of up to 74% are possible if a tolerance of 0,1 mm is used.

1 Inleiding

1.1 Situering

KUKA is een internationaal bedrijf dat zich voornamelijk focust op industriële robots en automatiseringssystemen. Deze masterproef gaat door in de vestiging in Houthalen, KUKA Automatisering + Robots N.V. Hier focust men zich vooral op het opbouwen van lijnen en het verder ontwikkelen van nieuwe technologieën, waaronder frezen met industriële robots.

Bedrijven gebruiken steeds meer industriële robots om repetitieve, eenvoudige, vuile of gevaarlijke arbeid van werknemers te automatiseren. Dankzij de vele mogelijke eindeffectors kunnen robots veel verschillende taken uitvoeren zoals beladen en ontladen van machines, pallettiseren, assembleren, verfspuiten, lassen, frezen… Bovendien kunnen robotarmen het zware werk voor arbeiders verlichten of hen bijstaan. Zoek altijd naar de meest bondige formulering: als je hetzelfde kan zeggen met minder woorden, dan doe je dat.

Een andere industriële toepassing voor robots is verspanen, waarbij de robot overbodig materiaal verwijdert om een werkstuk zijn gewenste vorm te geven. Een robotarm is immers in staat om verschillende materialen te frezen, van kunststoffen tot hout of zachte metalen [1], met een nauwkeurigheid van ongeveer 0,4 mm. Het frezen gebeurt door het omzetten van een CAD/CAM-bestand naar een bestand met punten, dat de robot gebruikt om een pad te vormen. Er worden twee verschillende bestanden aangemaakt. Het eerste bestand dient om het voorwerp te ruwen, waarbij de frees een groot deel materiaal wegneemt, zonder heel nauwkeurig te zijn. Het tweede bestand bevat heel precieze punten, waarmee de robot een pad bepaalt om het werkstuk zeer nauwkeurig uit te frezen.

Een industriële robot krijgt voorrang op een CNC-machine onder bepaalde omstandigheden [2], bijvoorbeeld indien er grotere werkstukken bewerkt moeten worden. Zo kan de door KUKA ter beschikking gestelde robot werkstukken frezen met een lengte tot 25 meter [1], terwijl een CNC-machine met een werkoppervlak van deze afmetingen veel duurder is. Voordelen van een CNC-machine zijn dan weer de robuustheid en de nauwkeurigheid. High-end CNC-machines kunnen een nauwkeurigheid behalen van 20-50 micron, terwijl een industriële robotarm een nauwkeurigheid van 100-200 micron kan behalen. Door de trillingen die ontstaan tijdens het frezen verandert de nauwkeurigheid naar 0,4 mm. Maar voor de meeste toepassingen is deze nauwkeurigheid meer dan voldoende. Onduidelijk of “de meeste toepassingen” nu enkel verwijst naar de “grotere werkstukken” of niet. Onduidelijk of voor kleinere werkstukken met lage gewenste nauwkeurigheid een CNC-machine interessanter is dan een robot of niet.

1.2 Probleemstelling

Het frezen met behulp van een robotarm is een tamelijk traag proces dat een hoge nauwkeurig vereist. Het genereren van extra punten zal de nauwkeurigheid van de baan verhogen. De procestijd zal hierdoor echter toenemen doordat de robot meer bewerkingen moet doen. Een langere procestijd zorgt niet alleen voor een hogere productiekost, maar ook voor een vermindering van het aantal gefreesde producten, wat bijgevolg de omzet doet dalen. Verder kan een gelijkaardig principe ook toegepast worden om producten te 3D-printen. Ook is hier een afweging tussen snelheid en nauwkeurigheid van belang. Het doel van KUKA is om software te ontwikkelen die er voor zorgt dat de verwerkingstijd vermindert waarbij de nauwkeurigheid binnen een gegeven tolerantie blijft.

De onderzoeksvraag luidt dus als volgt: *Hoe kan het freesproces versneld worden met behoud van de nauwkeurigheid binnen een gegeven tolerantie?*

1.3 Doelstellingen

De hoofddoelstelling van deze masterproef is de freesrobot sneller laten werken waarbij de nauwkeurigheid binnen de gegeven tolerantie blijft. Als eerste zal er getracht worden een programma te ontwerpen dat punten filtert aan de hand van een algoritme. Hiervoor moet er eerst een vergelijkende voorstudie uitgevoerd worden om te bepalen welk algoritme het meest geschikt is voor deze toepassing. Aan het algoritme moeten parameters meegegeven worden die de nauwkeurigheid van het filteren bepalen. Vervolgens wordt het programma getest waarbij de robot enkel lineaire bewegingen maakt. Indien het algoritme te veel punten verwijderd heeft of indien de nauwkeurigheid op bepaalde plaatsen niet hoog genoeg is, moet het programma punten toevoegen om de nauwkeurigheid te verbeteren. Wanneer het programma volledig werkt, is de volgende doelstelling om het algoritme aan te passen waarbij de robot gebruik maakt van spline-bewegingen. Spline-bewegingen hebben enkele voordelen ten opzichte van lineaire bewegingen, deze worden verder in de masterproef besproken. Door het toevoegen van punten die in het pad liggen, kunnen de spline-bewegingen beïnvloed worden. Er zal dus eerst onderzocht moeten worden hoe bepaalde punten het pad beïnvloeden.

Naast het filteren van punten op basis van de x-, y- en z-coördinaten, moet er ook nog rekening gehouden worden met de oriëntatie van de eindeffector. De eindeffector mag tussen 2 punten niet van oriëntatie veranderen, aangezien dit onvoorspelbare bewegingen kan opleveren. Hierdoor kan de eindeffector tegen het werkstuk of andere voorwerpen in de omgeving botsen met eventuele schade tot gevolg. Een optionele doelstelling is om het algoritme in 3D te laten werken, dit wil zeggen dat het algoritme niet alleen naar punten kijkt die op dezelfde hoogte liggen, maar ook naar punten die hoger of lager liggen. Op die manier kan de nauwkeurigheid ook in 3D binnen de tolerantie blijven.

1.4 Methode

Om de robot te kunnen programmeren en besturen is er eerst enige basiskennis vereist over de werking van de robot, de veiligheid en de bediening aan de hand van een touchscreen paneel. Hiervoor voorziet KUKA een 5-daagse opleiding. Vervolgens wordt er nog een tweede opleiding voorzien die zich meer verdiept in het frezen van producten met behulp van een robotarm. Hierna zal een literatuurstudie uitgevoerd worden, waarbij er meer inzicht over de onderzochte of reeds bestaande oplossingen verworven wordt.

Aan de hand van de tot dan toe verworven informatie wordt er een vergelijkende voorstudie uitgevoerd om het meest geschikte algoritme te bepalen. Vervolgens wordt dit algoritme geïmplementeerd en wordt het programma verder uitgewerkt. Hierna wordt het algoritme uitgebreid zodat het programma rekening houdt met de oriëntatie van de eindeffector. Ook moet het programma punten kunnen toevoegen indien de nauwkeurigheid niet hoog genoeg is. Het programma zal de minimale theoretische tijd om een stuk te frezen berekenen. Dit gebeurt aan de hand van de lengte van het pad en de maximale snelheid die toegelaten is om dat materiaal te frezen. Ten slotte zal er bestudeerd worden hoe splines werken en zal de code indien nodig aangepast worden.

Deze incrementele aanpak laat toe elke stap uitvoerig te testen of het programma naar behoren functioneert alvorens naar de volgende stap te gaan. Eerst zal er in 2D getest worden door eerst een figuur te tekenen met behulp van de gegeven punten. Daarna wordt dezelfde figuur getekend met behulp van de gefilterde punten. Hierdoor kan men de tijdwinst meten en de nauwkeurigheid vergelijken, deze tijdswinst zal voor elke figuur anders zijn. Het tekenen gebeurt door een pen te monteren aan de freeskop. Vervolgens zal er overgeschakeld worden naar 3D werkstukken. Door deze procedure echter voor meerdere werkstukken toe te passen, zal men een gemiddelde relatieve tijdswinst bekomen die vrij representatief is voor het merendeel van de werkstukken.

Figuur 1 toont de robotarm met een freeskop als eindeffector die KUKA voorziet. Deze robotopstelling zal toelaten het geschreven programma te testen op verschillende materialen waaronder kunststoffen, hout en zachte metalen zoals aluminium. Door te testen op de zachte kunststoffen zal een programmatiefout geen zware gevolgen hebben. Vervolgens zal een visuele controle bepalen of de nauwkeurigheid en snelheid voldoen aan de eisen. De code zal volledig in C# geschreven worden met behulp van Visual Studio.

Figuur 1: KUKA robotarm met een freeskop als eindeffector [3]

1.5 Vooruitblik

De literatuurstudie in hoofdstuk twee bespreekt een kort onderzoek over reeds bestaande filteralgoritmes voor andere toepassingen. Hierna wordt verder besproken hoe de fout die ontstaat door het filteren berekend kan worden. Als laatste bespreekt hoofdstuk twee kort de werking van de robot en het softwarepakket waarvan de robot gebruik maakt om te frezen.

Hoofdstuk drie bespreekt de selectie van het algoritme. De algoritmes die besproken werden in hoofdstuk twee zullen hier getest en vergeleken worden op basis van drie criteria. Op basis van deze testen zal een algoritme gekozen worden dat verder gebruikt wordt voor het filteren van punten.

Hoofdstuk vier bespreekt de volledige code voor het filteren van punten. Dit gaat van het inlezen van de punten tot het filteren en foutberekeningen op het filteren en ten slotte tot het uitschrijven van de gefilterde punten.

Hoofdstuk vijf bespreekt de resultaten van de testen die bekomen zijn indien de robot gebruik maakt van een lineaire beweging met luswerking.

Hoofdstuk zes bespreekt de resultaten van de testen die bekomen zijn indien de robot gebruik maakt van een spline-beweging.

Hoofdstuk zeven bespreekt het algemeen besluit dat geconcludeerd wordt uit de voorgaande testen.

Ten slotte wordt het toekomstig werk dat nog moet gebeuren voor het verder optimaliseren van het filteralgoritme en het implementeren van de spline functie in de robot besproken.

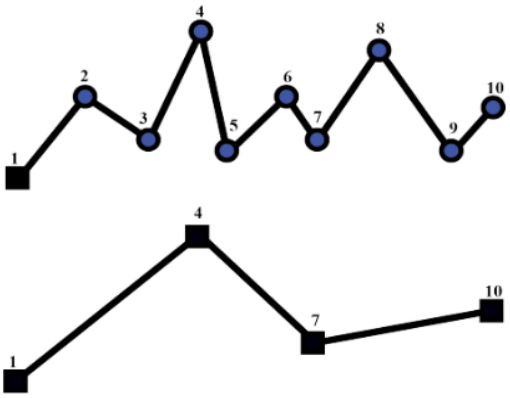
2 Literatuurstudie

2.1 Filteralgoritmes

De hoofdzaak van deze masterproef is het filteren van punten uit een pad dat gevolgd wordt door de industriële robot. Er is echter weinig informatie te vinden over het filteren van paden bij industriële robots. Andere toepassingen die algoritmes gebruiken om punten uit een lijn of pad filteren zijn mobiele robots of geografische toepassingen. De algoritmes die hier gebruikt worden kunnen, mits enige aanpassing, ook gebruikt worden voor industriële robots. Hieronder volgt een overzicht van de meest gebruikte algoritmes.

2.1.1 Nth-punt

Het Nth-punt algoritme is een eenvoudig O(n)-algoritme. Hierbij wordt het eerste, het laatste en elk n-de punt behouden zoals weergegeven in Figuur 2. Het voordeel van het Nth-punt algoritme is dat het zeer snel en eenvoudig is. Echter zorgt de eenvoud van het algoritme ook voor grote afwijkingen tussen het originele pad en het nieuwe pad [4], hierdoor komt dit algoritme niet in aanmerking voor de toepassing van deze masterproef.

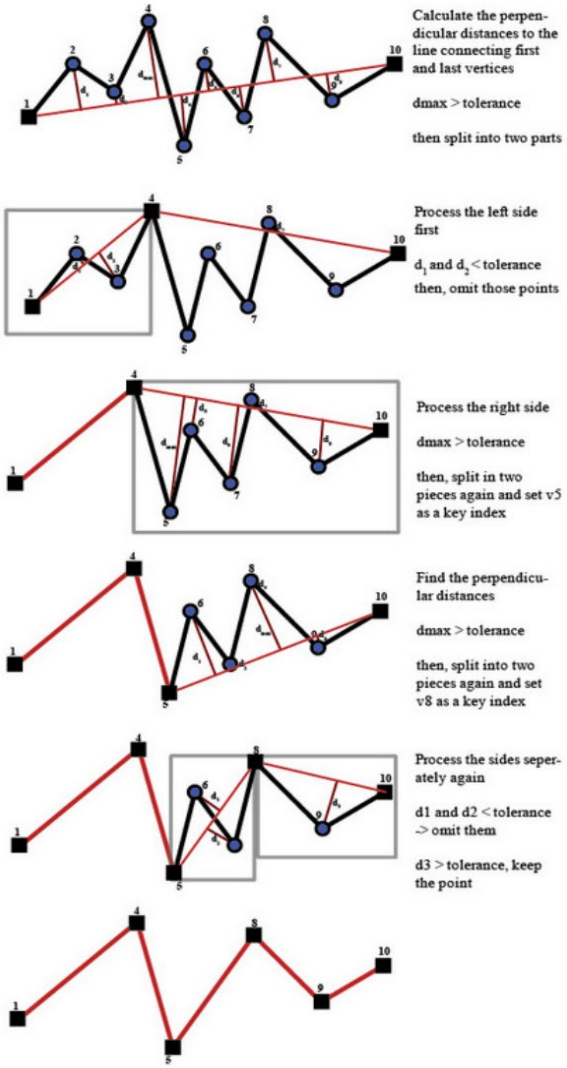


Figuur 2: Nth-punt algoritme met n=3 [4, p. 6]

2.1.2 Douglas-Peucker

Het Douglas-Peucker-algoritme probeert de algemene structuur van het pad te behouden door een tolerantie factor te gebruiken. Het eerste en het laatste punt van de *dataset* worden behouden. Tussen deze twee punten wordt vervolgens een lijn getrokken waarna van alle tussenliggende punten de loodrechte afstand tot deze lijn bepaald wordt. Indien het punt met de grootste loodrechte afstand tot de lijn binnen de tolerantie ligt, dan moet dit punt, samen met alle andere tussenliggende punten verwijderd worden. Indien de loodrechte afstand echter groter is dan de tolerantie, dan blijft ook dit punt behouden. Hierna worden er 2 nieuwe lijnen getekend tussen het eerste punt en het nieuwe punt en tussen het nieuwe punt en het laatste punt. Opnieuw zal dan van elk tussenliggend punt de loodrechte afstand bepaald worden tot deze lijnen en zal er gekeken worden of het punt met de grootste afstand verwijderd mag worden samen met alle andere punten, of dat het algoritme dit punt moet behouden. De vorige stappen worden herhaald tot alle punten ofwel behouden of verwijderd zijn. Deze stappen zijn weergegeven in Figuur 3 [4].

Het voordeel van het Douglas-Peucker-algoritme is dat de algemene structuur van het pad behouden blijft. De tolerantie die meegegeven wordt bepaald hoe sterk het algoritme van de algemene structuur mag afwijken. Voor elk punt zal echter de loodrechte afstand minstens één, maar meestal meerdere keren berekend moeten worden, hierdoor zal het algoritme vertragen. Indien het punt met de grootste loodrechte afstand gekend is en verwijderd mag worden, dan kunnen alle tussenliggende punten in één keer mee verwijderd worden [4].

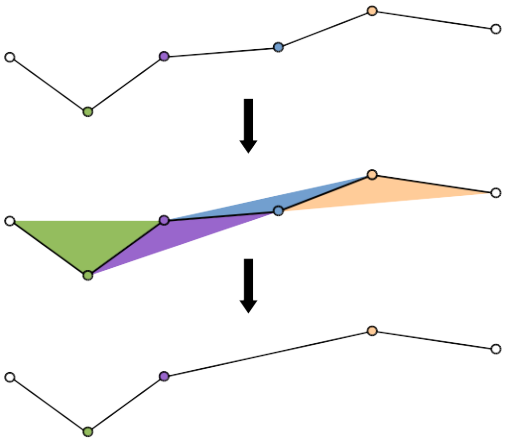


Figuur 3: Douglas-Peucker-algoritme [4, p. 12]

2.1.3 Visvalingam

Het Visvalingam algoritme maakt gebruik van driehoeken om te bepalen of punten al dan niet gefilterd mogen worden. Figuur 4 geeft de werking van het algoritme weer. Als eerste zal de oppervlakte van een driehoek tussen elke drie opeenvolgende punten bepaald worden. Het middelpunt van de driehoek met de kleinste oppervlakte zal verwijderd worden indien de oppervlakte kleiner is dan de ingestelde tolerantie. Hierna zullen opnieuw de oppervlaktes tussen elke drie opeenvolgende punten bepaald worden en zal het middelpunt van de kleinste driehoek verwijderd worden. Dit proces zal herhaald worden tot de kleinste driehoek een grotere oppervlakte heeft dan de ingestelde tolerantie [5].

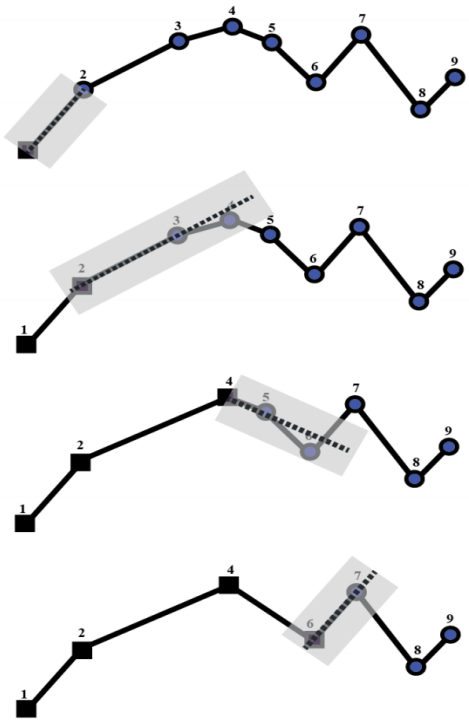
Visvalingam heeft als voordeel dat het relatief een zeer goed resultaat behaalt indien er een grote tolerantie gekozen wordt. De algemene structuur zal beter herkenbaar blijven dan bij andere algoritmes [6]. Het nadeel van dit algoritme is echter dat het langzaam is. Verder behaalt Douglas-Peucker ook een beter resultaat bij kleine toleranties. Voor deze masterproef worden enkel kleine toleranties gebruikt, hierdoor zal het Douglas-Peucker-algoritme de voorkeur krijgen op het Visvalingam algoritme [5].



Figuur 4: Visvalingam algoritme [7]

2.1.4 Reumann-Witkam

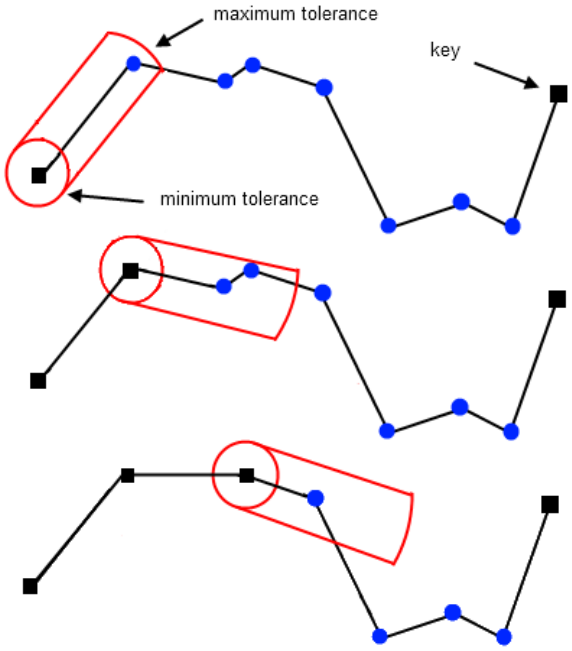
Het Reumann-Witkam algoritme bepaalt, zoals weergegeven in Figuur 5, een lijn tussen de eerste twee punten, deze punten blijven behouden. De lijn wordt doorgetrokken tot aan de loodrechte positie van het volgende punt. Indien dit punt niet binnen het tolerantiegebied ligt, dan moet het vorige punt behouden blijven. Indien het punt wel binnen het tolerantiegebied ligt, dan wordt de lijn verder doorgetrokken tot aan het volgende punt en wordt er opnieuw gekeken of dit punt al dan niet binnen de tolerantie ligt. Hierna wordt er opnieuw een lijn bepaald tussen de twee volgende punten, op deze manier zal het algoritme herhaald worden tot alle punten doorlopen zijn. De hele dataset kan meerdere keren doorlopen worden om extra punten te filteren [4].



Figuur 5: Reumann-Witkam algoritme [4, p. 8]

2.1.5 Opheim

Het Opheim algoritme heeft veel gelijkenissen met het Reumann-Witkam algoritme. Allereerst wordt er een zoekgebied gedefinieerd. Dit gebeurt aan de hand van een minimale en maximale tolerantie en twee rechten. Rond het eerste punt worden twee cirkels bepaald waarvan de stralen gelijk zijn aan de minimale en maximale tolerantie. Vervolgens worden er twee rechten getekend die evenwijdig zijn met de rechte door het eerste en volgende punt. De afstand tussen de parallelle rechten wordt bepaald door de diameter van de kleine cirkel. Hierna worden alle punten die in het zoekgebied liggen verwijderd, behalve het laatste punt in het zoekgebied. Indien er zich maar één punt in het zoekgebied bevindt, wordt dit punt altijd behouden. Vervolgens wordt het principe toegepast op het volgende punt uit de dataset, dit wordt herhaald tot alle punten zijn doorlopen. Figuur 6 visualiseert het algoritme [8].

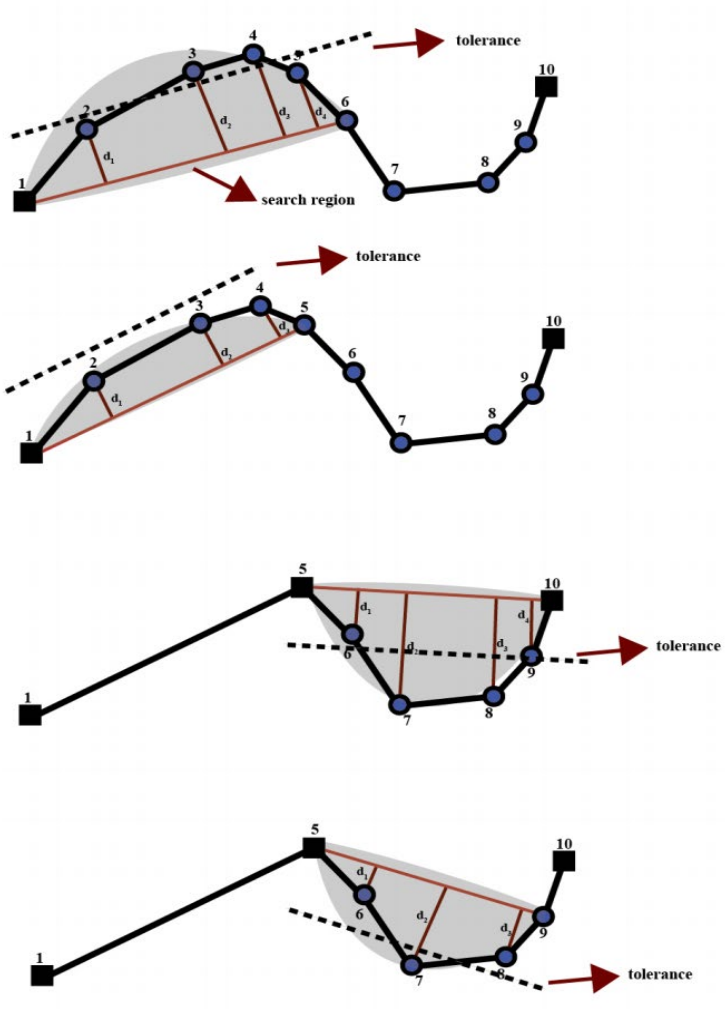


Figuur 6: Opheim algoritme [8, p. 31]

2.1.6 Lang

Bij het Lang algoritme moet er zowel een zoekgebied als een tolerantie meegegeven worden. Er wordt een lijn bepaald tussen punt 1 en punt 1+n waarbij n de grootte van het zoekgebied is. Het tolerantiegebied ligt parallel aan de lijn met de tolerantie als afstand. Indien er van de tussenliggende punten een punt is dat buiten dit tolerantiegebied valt, dan wordt het zoekgebied verminderd met één. Wanneer alle punten binnen het zoekgebied vallen, dan mogen deze allemaal verwijderd worden en wordt er verder gegaan [4]. Figuur 7 toont de werking van het Lang algoritme.

Bij dit algoritme is het zeer belangrijk dat er een goede schatting gemaakt wordt van het zoekgebied, best kan dit iets te groot genomen worden. Indien het zoekgebied te klein genomen wordt, dan zullen er meer punten dan nodig behouden blijven [4].

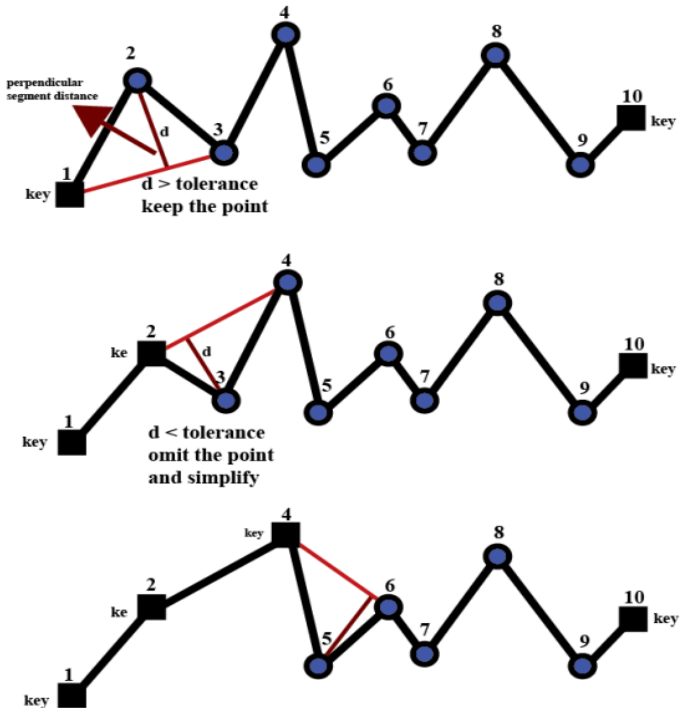


Figuur 7: Lang algoritme [4, p. 12]

2.1.7 Perpendicular distance

Het perpendicular distance algoritme lijkt in veel opzichten op het Douglas-Peucker-algoritme. Ook hier wordt er gebruik gemaakt van de loodrechte afstand om te bepalen of een punt verwijderd wordt of behouden blijft. Het eerste en het laatste punt blijven behouden. Vervolgens wordt er een lijn bepaald tussen het eerste en het derde punt. Indien de loodrechte afstand van het tweede punt tot deze lijn groter is dan de meegegeven tolerantie, dan zal het punt behouden blijven. Indien de afstand echter kleiner is, dan mag het punt verwijderd worden. Hierna gaat het algoritme verder naar de volgende punten zoals weergegeven in Figuur 8 [4].

Het nadeel aan dit algoritme is dat de lijst met punten meerdere keren doorlopen moet worden om een groot aantal punten te verwijderen. Hierdoor is het efficiënter om het Douglas-Peucker-algoritme te gebruiken, bovendien verwijdert Douglas-Peucker meer punten indien dezelfde tolerantie gebruikt wordt [4].



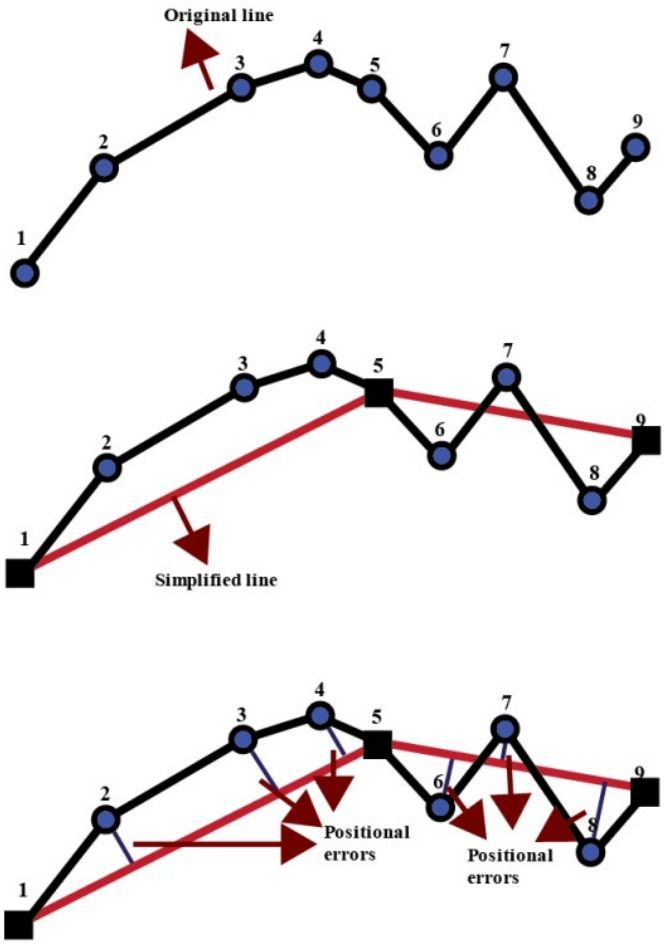
Figuur 8: Perpendicular distance algoritme [4, p. 7]

2.2 Foutimplementatie

Indien een van bovenstaande filteralgoritmes wordt toegepast op de originele baan van de robot, ontstaat er een vereenvoudigde baan. Deze baan heeft een bepaalde vervorming ten opzichte van de originele baan. Om de vervorming weer te geven, kan de fout op basis van twee principes berekend worden. Het eerste principe werkt op basis van loodrechte afstanden. Het tweede principe maakt gebruik van oppervlaktes [4].

2.2.1 Positiefout

De positiefout berekent van elk punt de loodrechte afstand tot de vereenvoudigde baan. De som van deze loodrechte afstanden geeft een idee van de totale vervorming van de vereenvoudigde baan. Figuur 9 geeft de methode van de positiefout weer [4].



Figuur 9: Positiefout [4, p. 16]

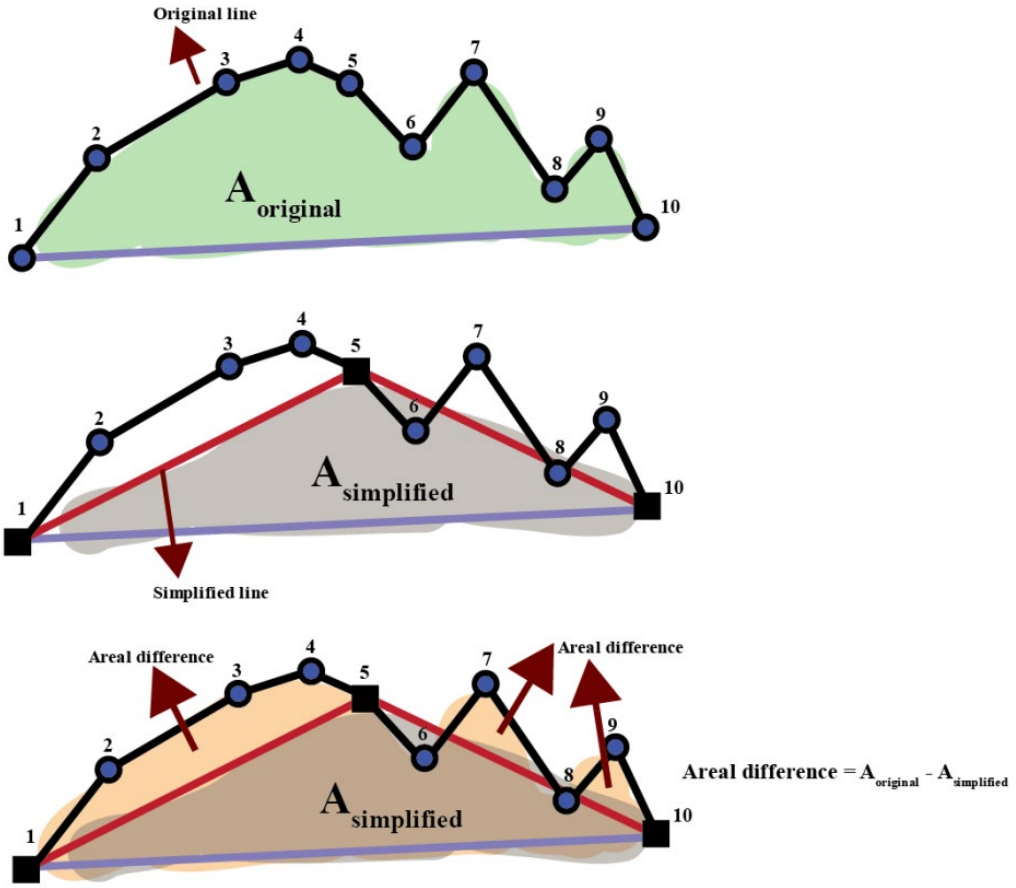
Het principe van de positiefout kan ook als volgt in een formule weergegeven worden:

$$PES = \sum_{i=0}^{nv} vls(i) \tag{1}$$

Hierbij is nv het aantal punten dat de originele baan beschrijft. *Vls* is de lengte van de loodrechte afstand in mm van een punt van de originele baan tot de vereenvoudigde baan [4].

2.2.2 Oppervlaktefout

De positiefout geeft inzicht in de verplaatsing van de punten, maar geeft minder inzicht in de geometrische vervorming van de vereenvoudigde baan. Om een goed idee te krijgen van de geometrische vervorming van de baan, wordt er gebruik gemaakt van de oppervlaktefout. Figuur 10 visualiseert dit principe [4].



Figuur 10: Oppervlaktefout [4, p. 17]

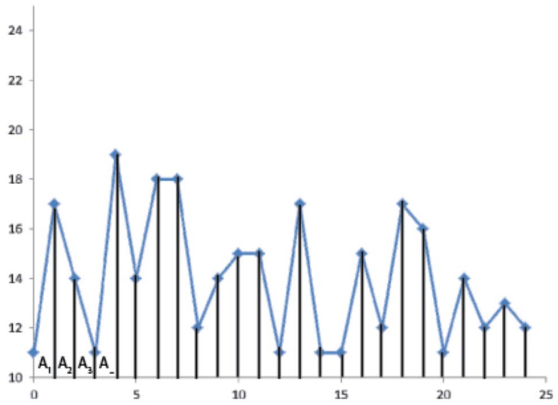
Eerst wordt de oppervlakte van de baan berekend ten opzichte van de rechte die het eind- en beginpunt met elkaar verbindt. Vervolgens wordt de oppervlakte van de vereenvoudigde baan op exact dezelfde manier berekend. De totale oppervlaktefout is de oppervlakte van de originele baan min de oppervlakte van de vereenvoudigde baan. Het berekenen van de oppervlakte onder een baan kan op verschillende manieren bepaald worden. Vervolgens worden er twee methodes weergegeven namelijk, de trapeziumregel en de regel van Simpson [4].

2.2.2.1 Trapeziumregel

De oppervlakte onder een functie kan berekend worden aan de hand van een integraal. De trapeziumregel is een benadering van een integraal door middel van oppervlaktes van trapeziums te bereken in een interval [a, b]. De formule is als volgt [4].

$$\int_a^b f(x)dx \approx (b-a) \cdot \frac{f(a)+f(b)}{2} \tag{2}$$

Onderstaande Figuur 11 visualiseert de trapeziumregel.



Figuur 11: Trapeziumregel [4, p. 18]

2.2.2.2 Regel van Simpson

De regel van Simpson is een andere manier voor het benaderen van een integraal van een functie f door gebruik te maken van kwadratische veeltermen (in plaats van rechte lijnen). De regel van Simpson luidt als volgt:

$$A = \frac{1}{2}h(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + 4y_5 + \cdots + 4y_{n-1} + y_n) \tag{3}$$

Hierbij is h gelijk aan de breedte van een segment $\frac{eindpunt-beginpunt}{aantal\ punten}$. Indien een hoge nauwkeurigheid gewenst wordt, zal n zeer groot zijn [4].

2.3 Foutanalyse

Om te bepalen welk algoritme het meest geschikt is voor deze masterproef, worden de algoritmes met elkaar vergeleken. Bepaalde algoritmes worden zeer weinig of niet gebruikt, zoals de nth punt omdat deze belangrijke punten kan verwijderen. Deze literatuurstudie focust zich voornamelijk op het Douglas-Peucker en het perpendicular distance algoritme.

Om het Douglas-Peucker en het perpendicular distance algoritmes met elkaar te vergelijken, maakt men gebruik van de positiefout en wordt hiervan het gemiddelde bepaald [9]. De positiefout is niet de beste manier om de exacte vervorming te bepalen, maar geeft een goede indicatie tussen het verschil in nauwkeurigheid van de verschillende algoritmes. Omdat beide algoritmes gebruik maken van een *threshold* parameter die de nauwkeurigheid bepaalt, is het niet evident om beiden te vergelijken. Om de algoritmes toch te kunnen vergelijken, past men ze toe op veertien verschillende banen met verschillende thresholds. Tabel 1 en Tabel 2 geven hiervan het resultaat weer.

Tabel 1: Resultaten testen met Douglas-Peucker en perpendicular distance algoritme met verschillende parameters [9, p. 67]

Line code		L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8	L_9	L_{10}	L_{11}	L_{12}	L_{13}	L_{14}	Simplification threshold $\varepsilon(m)$	Average rate of line simplification(%)
Points before simplification		26	41	41	41	51	46	36	66	66	56	51	36	31	21		
Points after simplification	Step algorithm	19	23	29	28	27	31	31	40	42	36	29	18	15	11	1.500	37.8
		17	22	24	25	25	26	25	35	37	32	24	17	14	11	1.700	44.9
		16	20	21	22	23	22	23	30	31	26	21	16	12	11	2.000	50.9
		12	18	19	19	21	20	20	28	27	24	20	13	10	10	2.500	56.8
		11	15	16	16	17	18	18	25	24	20	16	11	9	8	3.000	63.0
	Line segment filtration algorithm	19	24	28	28	24	30	31	37	39	33	25	14	11	12	1.500	41.3
		19	22	25	26	24	28	29	37	36	31	23	13	10	11	1.530	44.9
		15	19	22	23	19	21	24	26	25	24	16	11	7	10	1.800	56.0
		14	18	18	20	16	17	22	23	21	19	14	10	5	9	2.000	61.8
		10	15	17	17	13	16	19	19	18	17	13	9	4	7	2.200	67.6
	Perpendicular distance algorithm	17	22	18	20	22	30	25	36	40	37	27	22	15	17	0.050	41.8
		15	20	17	19	22	29	24	35	37	36	25	22	14	17	0.055	44.5
		11	16	12	16	14	21	22	26	28	30	22	15	12	16	0.075	55.8
		10	11	10	12	13	17	19	23	24	23	16	11	8	13	0.100	64.6
		9	11	9	10	12	14	18	19	20	19	14	11	7	10	0.120	69.1
	Douglas-Peucker algorithm	18	23	21	24	28	32	26	41	45	40	33	26	17	18	0.050	34.9
		16	21	16	19	22	25	26	35	38	33	28	19	16	16	0.070	44.8
		11	17	13	16	19	23	21	29	30	28	21	16	10	15	0.100	55.1
		10	11	13	15	14	15	18	22	22	24	18	12	10	11	0.150	63.8
		9	10	11	11	14	15	16	20	22	21	15	11	10	9	0.200	67.6

Hier is de *average rate of line simplification* het gemiddelde van de verhouding van het aantal gefilterde punten, bij een bepaalde threshold en baan, op het aantal oorspronkelijke punten. Vervolgens wordt er bij een gelijk percentage van de average rate of line simplification de gemiddelde positiefout bepaald [9].

Tabel 2: Samenvattende resultaten testen met Douglas-Peucker en perpendicular distance algoritme met verschillende parameters [9, p. 67]

Δ_i (m) Line code and algorithm \ Point code		P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	mean error (m)
L_1	Step algorithm	0.018	0.056	0.087	0.058	0.028	0.016	0.014	0.013	0.044				±0.044
	Line segment filtration algorithm	0.012	0.041	0.051	0.078	0.082	0.069	0.062	0.039	0.031	0.017	0.014	0.016	±0.049
	Perpendicular distance algorithm	0.016	0.056	0.041	0.048	0.061	0.047	0.037	0.059	0.051	0.047	0.045		±0.048
	Douglas-Peucker algorithm	0.046	0.060	0.054	0.027	0.014	0.029	0.024	0.021	0.048	0.021			±0.037
L_7	Step algorithm	0.210	0.119	0.005	0.011	0.148	0.134	0.038	0.016	0.109	0.125	0.063		±0.109
	Line segment filtration algorithm	0.136	0.161	0.133	0.042	0.018	0.098	0.150	0.138	0.077	0.053	0.082	0.066	±0.106
	Perpendicular distance algorithm	0.026	0.101	0.097	0.079	0.052	0.028	0.009	0.028	0.065	0.059	0.045	0.030	±0.059
	Douglas-Peucker algorithm	0.027	0.018	0.024	0.026	0.028	0.010	0.029	0.013	0.023	0.032			±0.024
L_{14}	Step algorithm	0.027	0.088	0.138	0.081	0.031	0.035	0.101	0.215	0.112	0.031			±0.103
	Line segment filtration algorithm	0.016	0.064	0.079	0.141	0.135	0.105	0.046	0.224	0.334	0.242	0.097	0.078	±0.157
	Perpendicular distance algorithm	0.025	0.055	0.018	0.028									±0.034
	Douglas-Peucker algorithm	0.027	0.043	0.018	0.041	0.030								±0.033

Deze resultaten zijn een goede aanwijzing dat het Douglas-Peucker-algoritme het meeste nauwkeurige algoritme is. Het nadeel is dat het een relatief traag algoritme is ten opzichte van het perpendicular distance algoritme [9]. Op basis van deze bevindingen wordt er een diepgaander onderzoek gedaan naar Douglas-Peucker, vooral gefocust op de beperkingen ervan. In hoofdstuk 4 zullen alle algoritmes uitgewerkt worden om een betere vergelijking te krijgen.

2.4 Beperkingen Douglas-Peucker

Zoals elk algoritme heeft ook het Douglas-Peucker-algoritme voor- en nadelen. De volgende paragrafen beschrijven situaties waarbij het Douglas-Peucker-algoritme faalt of fouten maakt.

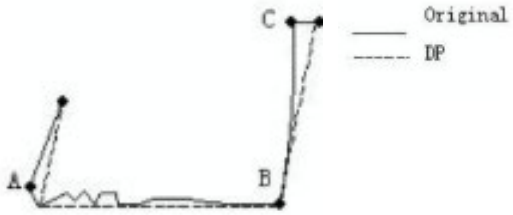
2.4.1 Self-intersection

Origineel heeft de baan geen snijpunten met zichzelf. Door het filteren met behulp van het Douglas-Peucker-algoritme kan het voorkomen dat de gefilterde baan zichzelf wel snijdt. Dit is echter niet gewenst indien er wordt gefreesd. Echter is dit enkel het geval indien de tolerantie van het algoritme een grote waarde heeft. Dit fenomeen kan vermeden worden door de tolerantie kleiner te maken of door het algoritme aan te passen. Ying Shen en Guo Renzhong hebben hier in [10] een oplossing voor bedacht. Deze zal echter niet verder besproken worden, omdat in de toepassing van deze masterproef dit fenomeen niet zal voorkomen doordat er met een zeer kleine tolerantie gewerkt wordt. Figuur 12 geeft een voorbeeld van *self-intersection*.

Figuur 12: Self-intersection [10, p. 3]

2.4.2 Keypoint removing

Door het filteren van de punten kunnen er soms belangrijke punten verwijderd worden. Dit zijn punten die cruciaal zijn voor de vorm en indien ze verwijderd worden zal er een hoge vervorming van de originele baan plaatsvinden. Deze cruciale punten worden ook wel *keypoints* genoemd. Figuur 13 geeft een situatie weer waarin het algoritme een keypoint verwijdert [11].



Figuur 13: Keypoint removing [11, p. 2]

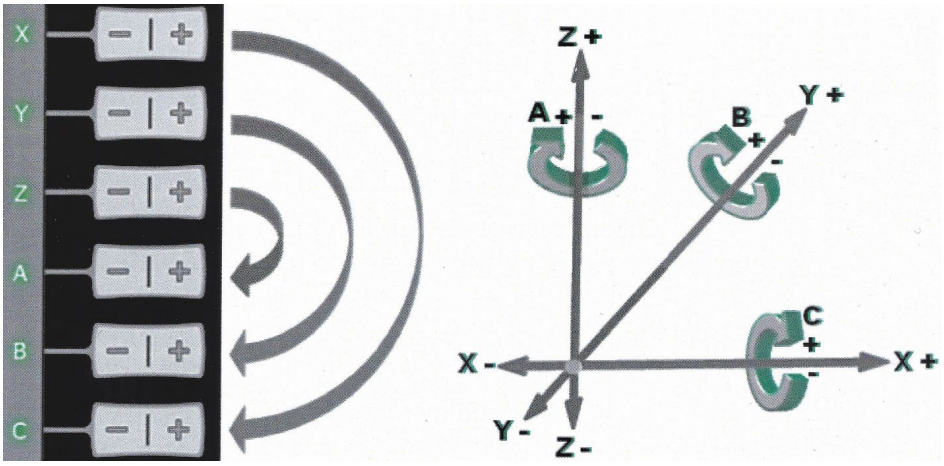
Punten die de originele baan beschrijven worden als een vet punt weergegeven. De originele baan is weergeven in volle lijn en de vereenvoudigde baan in stippellijn. Het algoritme start met een rechte te bepalen door het begin- en eindpunt. Het punt dat het verste van deze rechte ligt is B. Vervolgens wordt er een rechte door punt B en het eindpunt bepaald, indien de tolerantie groter is dan de afstand van C tot de recht, wordt het punt C verwijderd [11]. Net zoals self-intersection, zal ook keypoint removing niet van toepassing zijn op deze masterproef aangezien er gewerkt wordt met zeer kleine toleranties. Hierdoor is de kans dat een keypoint verwijderd wordt zeer klein.

2.5 Robotwerking

2.5.1 Algemene robotwerking

Een industriële robot kan in verschillende coördinatenstelsels geprogrammeerd worden [12], [13]. Als eerste is programmatie mogelijk in cartesische coördinaten, deze kunnen opgedeeld worden in wereld-, *base*- en *toolcoördinaten*. Bij wereldcoördinaten ligt de oorsprong van het assenstelsel meestal in de voet van de robot. Basecoördinaten zijn vrij definieerbaar, dit wil zeggen dat dit assenstelsel vrij in de wereld gekozen kan worden en dus ook op het werkstuk. Voor deze masterproef zal er gewerkt worden met basecoördinatensystemen. Toolcoördinatensystemen hebben hun oorsprong in de TCP. Naast deze cartesische coördinatensystemen kan de robot ook geprogrammeerd worden in robotcoördinaten. Hierbij kan elke joint van de robot individueel bewogen worden.

In deze cartesische coördinatensystemen kan op twee verschillende manier bewogen worden [12]. De eerst mogelijke beweging is een translatie. Hierbij beweegt de TCP in de x-, y- of z-richting. Daarnaast is er ook een rotatie mogelijk. Hierbij blijft de TCP stilstaan op dezelfde x-, y- en z-coördinaat, maar zal de tool zelf draaien of zwenken rondom zijn TCP. De tool kan rond 3 verschillende hoeken draaien, dit zijn de Euler hoeken A, B en C. Hoek A komt overeen met een rotatie rond de z-as, hoek B komt overeen met een rotatie rond de y-as en hoek C komt overeen met een rotatie rond de x-as. De positieve draairichting kan bepaald worden met behulp van de rechterhandregel. Dit is weergegeven in Figuur 14.

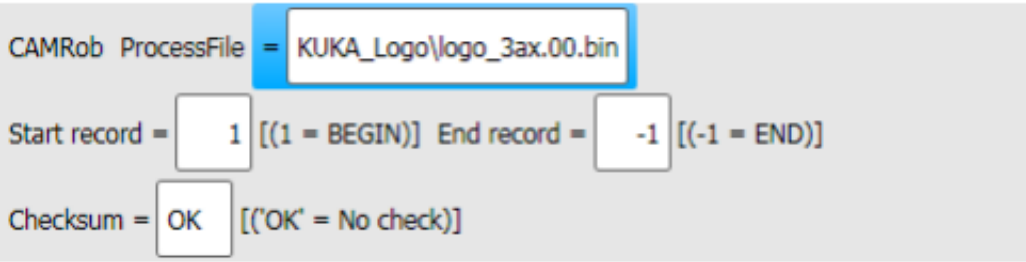


Figuur 14: Rotatie rond de verschillende assen van het cartesische assenstelsel [12, p. 37]

Door middel van inverse kinematica kan een industriële robot bewegingen uitvoeren in een cartesisch coördinatensysteem [13]. De robot zal deze cartesische coördinaten omrekenen tot asspecifieke coördinaten. Een industriële robot kan twee verschillende bewegingstypes uitvoeren in een cartesisch coördinatensysteem. Een eerste bewegingstype is de puntbeweging, hieronder vallen PTP bewegingen. Bij een PTP beweging moet enkel het eindpunt geprogrammeerd worden. De robot zal berekenen wat de stand van zijn verschillende assen moet zijn om dit eindpunt te bereiken, hierna zal de robot al zijn assen gelijktijdig bewegen tot deze in de eindpositie staan. Doordat de robot alle assen individueel zal bewegen wordt er geen rekening gehouden met het pad. Het voordeel van een PTP beweging is dat dit de snelst mogelijke beweging is die de robot kan uitvoeren tussen twee punten. Het tweede bewegingstype is een *contourbeweging*, bij contourbewegingen is de baan van het pad gespecifieerd in cartesische coördinaten. Mogelijke banen zijn: een rechte lijn, een cirkel en een *spline-beweging*. Een cirkelbeweging zal in deze masterproef niet verder besproken worden. De rechtlijnige beweging wordt verder besproken in hoofdstuk 2.5.2. De spline-beweging wordt verder besproken in hoofdstuk 2.5.3.

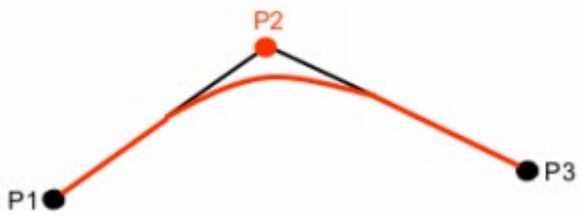
2.5.2 KUKA.CAMRob

KUKA.CAMRob is de software die op de freesrobot draait [14]. Alle punten waarlangs de robot moet frezen kunnen meegegeven worden in een *process file*. Deze process file kan een bestand zijn met CSV, BIN of TXT als extensie. Dit bestand bevat alle informatie die de robot nodig heeft om het freesproces te kunnen afwerken. Doordat niet elk freesproces hetzelfde is, zijn er verschillende parameters die meegegeven kunnen worden. Niet elke parameter moet verplicht meegegeven worden, hierdoor kan de opbouw van de process file anders zijn voor verschillende toepassingen. Enkele parameters zijn wel verplicht. Als eerste is een volgnummer nodig, beginnende bij 1. Dit volgnummer houdt bij hoeveel punten er zijn en op welke positie de robot zich op dat ogenblik bevindt in het bestand. Naast het volgnummer zijn ook nog de x-, y- en z-coördinaat van elk punt en de Euler hoeken A, B en C voor de verdraaiing van de *tool* verplicht. Naast deze zeven verplichte parameters kunnen er nog 26 andere parameters meegegeven worden. Enkele van deze parameters zijn: snelheid, versnelling, toolnummer, koeling, toerental van de spindel, spindel aan/uit en eventuele externe assen E1 tot E6. Deze process file kan vervolgens meegegeven worden aan de robot door middel van een *inline* formulier zoals weergegeven in Figuur 15.



Figuur 15: Inline formulier voor de process file command [14, p. 27]

Tijdens het uitvoeren van bovenstaande *command line*, zal de robot beginnen met frezen. De robot maakt hierbij een contourbeweging en meer bepaald lineaire bewegingen tussen twee opeenvolgende punten [14]. De robot zal een rechte baan volgen tussen de 2 punten, het eindpunt zal echter met luswerking benaderd worden. Dit wil zeggen dat de TCP van de robot niet precies door het punt zal gaan, maar de robot begint op een gegeven afstand van het punt zijn baan richting het volgende punt in te zetten. Hierdoor wordt de hoek tussen drie opeenvolgende punten afgerond, maar zal de robot zijn snelheid beter kunnen behouden. Er zal echter ook een kleine fout gemaakt worden die bij frezen niet gewenst is. Figuur 16 geeft een voorbeeld van luswerking weer.



Figuur 16: Luswerking bij een lineaire beweging van de robot [15, p. 357]

2.5.3 Spline-beweging

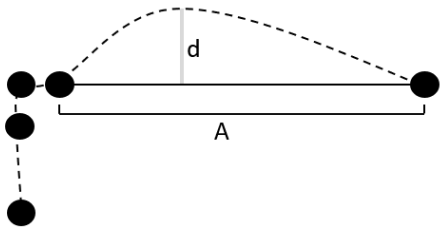
KUKA robots maken gebruik van een spline bestaande uit polynomen van de 5de orde. De literatuurstudie zal zich dus ook beperken tot dit specifiek type van spline. Een spline is een kromme bestaand uit verschillende polynomen, de hoogste graad van de polynomen is de graad van de spline [16]. Een spline is een functie die door elk gegeven punt zal gaan. Hierdoor kan de spline over een interval [a, b] onderverdeeld worden in deelintervallen $[x_{i-1}, x_i]$. Indien de spline bestaat uit k veeltermen van S_i polynomen in het interval [a, b], dan geldt:

$$S(x) = \begin{cases} S_0(x) & x \in [x_0, x_1] \\ S_1(x) & x \in [x_1, x_2] \\ \vdots & \vdots \\ s_{k-1}(x) & x \in [x_{k-1}, x_k] \end{cases} \quad (4)$$

Hierbij geldt: $a = x_0 < x_1 < \cdots < x_{k-1} < x_k = b$.

Verder zijn er nog extra randvoorwaarden nodig. De spline zal na toepassing van bovenstaande definitie immers niet continu zijn in ieder punt. Om toch een continue spline te bekomen moet in elk knooppunt $S_i(x_i)$ gelijk zijn aan $S_{i-1}(x_i)$. Om deze spline ook een vloeiende beweging te laten maken moeten ook de 1ste en 2de orde afgeleide van $S_i(x_i)$ en $S_{i-1}(x_i)$ gelijk zijn. Na het toepassen van deze randvoorwaarden blijven er bij hogere orde splines met een graad van drie of hoger, nog twee vrijheidsgraden over. Deze vrijheidsgraden kunnen weggewerkt worden door extra randvoorwaarden. Deze randvoorwaarden zijn de raaklijnen aan het begin en eindpunt van de spline. Indien deze raaklijnen zo gekozen worden dat de tweede afgeleide in het begin- en eindpunt gelijk zijn aan 0, dan wordt dit een natuurlijke spline genoemd.

Naast bovenstaande randvoorwaarden heeft de KUKA robot nog extra begrenzingen om te zorgen voor een vloeiende beweging. Zo zal de snelheid van de eindeffector zo constant mogelijk gehouden worden. De snelheid zal enkel veranderen indien één van volgende gevallen voorkomt in het pad: er korte bochten gemaakt worden, de eindeffector zich onder een grote hoek moet heroriënteren over een zeer korte afstand, de externe assen grote bewegingen moeten maken of de beweging zich voordoet in de buurt van singulariteiten [15]. De eindeffector kan ook volledig tot stilstand komen tijdens het uitvoeren van een spline-beweging. Dit zal voorkomen indien 2 punten dichter dan 0,1 mm van elkaar liggen. Bovendien mag de spline niet te hard afwijken van de lineaire beweging. Zo zal de spline maximaal de helft van de afstand tussen twee opeenvolgende punten mogen afwijken van de lineaire functie. Figuur 17 visualiseert de laatste randvoorwaarden. Hierbij is *A* de afstand tussen de twee opeenvolgende punten en *d* de maximale afwijking van de spline ten opzichte van de lineaire functie.



Figuur 17: Randvoorwaarden spline: maximale afwijking van de spline

Indien er gebruik gemaakt wordt van meerdere opeenvolgende spline-bewegingen is het aangewezen om een spline *block* te gebruiken. Een spline block begint met spline en eindigt met *endspline*. De juiste syntaxis voor het gebruik van een spline block is terug te vinden in Figuur 18.

```
SPLINE < WITH SysVar1 = Value1 <, SysVar2 = Value2, ... > >
Segment1
...
<SegmentN>
ENDSPLINE <C_SPL>
```

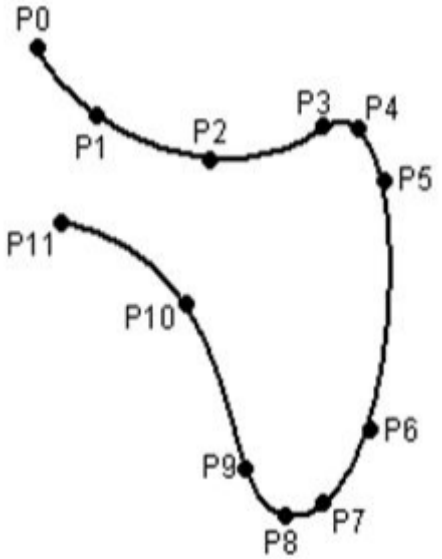
Figuur 18: Syntax voor het gebruik van een spline block [15, p. 469]

Op de plaats van de systeemvariabelen kunnen variabelen gedeclareerd worden die een invloed hebben op de robot. Enkele van deze variabelen zijn: de snelheid, de versnelling, de gebruikte tool, de gebruikte base,

de *jerk* etc. Op de plaats van de segmenten kunnen de bewegingen gedefinieerd worden. Hierbij is er een mogelijkheid tussen een SCIRC, SLIN of een SPL beweging. Voor deze masterproef wordt het gebruik gelimiteerd tot de SPL beweging. Een SPL beweging is een standaard spline-beweging. SCIRC en SLIN zijn complexere splines waarbij respectievelijk een perfecte cirkel of een perfect rechte lijn getekend wordt.

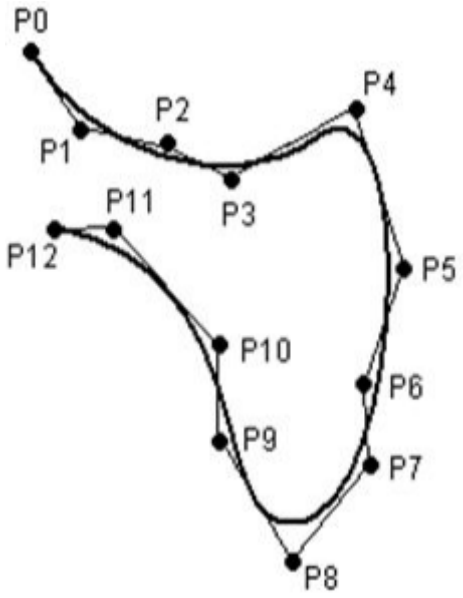
Figuur 19 geeft een voorbeeld van een spline door 12 punten , gemaakt door een KUKA robot. Het gebruik van splines heeft een aantal voordelen ten opzichte van lineaire bewegingen met luswerking, deze voordelen zijn [15]:

- Het pad wordt gegenereerd door punten die zich effectief op het pad bevinden. Hierdoor kan het pad makkelijk bepaald worden.
- Zoals hierboven uitgelegd, wordt de snelheid veel constanter op de geprogrammeerde snelheid gehouden.
- Het pad blijft altijd hetzelfde, ongeacht *override settings*, geprogrammeerde snelheid of versnelling. Dit is niet het geval bij lineaire bewegingen met luswerking.
- Cirkels en korte bochten worden met veel precisie gemaakt.



Figuur 19: Pad gevolgd door de robot gebruik makende van spline block [15, p. 362]

Figuur 20 geeft hetzelfde pad weer als in Figuur 19, deze keer wordt er echter gebruik gemaakt van een lineaire beweging met luswerking in plaats van een spline. Zoals te zien is op de figuur, gaat het pad deze keer niet door de punten. De punten moeten dus zo geprogrammeerd worden dat de eindeffector toch het gewenste pad volgt. Dit brengt extra moeilijkheden met zich mee tijdens het programmeren van de robot. Het pad zal niet constant blijven indien override settings, de snelheid of versnelling aangepast worden in de programmacode. Indien men hetzelfde pad wil behouden met andere parameters, zal men dus ook alle punten opnieuw moeten programmeren.

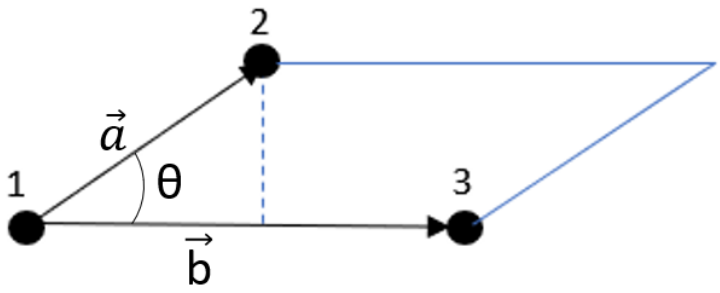


Figuur 20: Punten nodig om het pad uit Figuur 19 te verkrijgen gebruik makende van lineaire bewegingen met luswerking [15, p. 362]

3 Vergelijkende voorstudie

3.1 Foutimplementatie

Uit de literatuurstudie blijkt dat er verschillende algoritmes zijn om punten van paden te filteren. Om het meest geschikte algoritme te bepalen voor de toepassing van deze masterproef worden alle algoritmes met elkaar vergeleken. De vergelijking gebeurt op basis van drie criteria namelijk: de snelheid van het algoritme, de nauwkeurigheid van het gefilterde pad ten opzichte van het oorspronkelijk pad en de eenvoud van de in te stellen parameter. De snelheid van het algoritme wordt softwarematig bepaald door het plaatsen van *stopwatches* in de code. De nauwkeurigheid van het gefilterde pad kan op twee manieren bepaald worden. Door middel van de positiefout of de oppervlaktefout. Indien er meerdere punten achter elkaar worden gefilterd, moet er een oppervlakte berekend worden die beschreven wordt door minimaal vier punten. Dit is echter niet altijd mogelijk in een 3D-situatie. Daarom wordt de fout berekend aan de hand van de positiefout. Figuur 21 geeft een voorbeeld weer waarbij punt 2 gefilterd wordt.



Figuur 21: Berekening positiefout

In de code wordt dit geïmplementeerd door gebruik te maken van de *library* Vector3. Deze klasse maakt van twee punten een driedimensionale vector. Door het definiëren van twee vectoren, een vector *a* van punt 1 naar het gefilterde punt 2 en een vector *b* van punt 1 naar het overgebleven punt 3, kan het kruisproduct berekend worden.

$$Oppervlakte = \|\vec{a} \times \vec{b}\| = \|\vec{a}\| \cdot \|\vec{b}\| \cdot \sin\theta \tag{5}$$

De grootte van het kruisproduct van deze twee vectoren is gelijk aan de oppervlakte van de parallellogram, gevormd door de twee vectoren. De loodrechte afstand oftewel de fout is gelijk aan de oppervlakte gedeeld door de grootte van de vector b.

$$fout = \frac{\|\vec{a} \times \vec{b}\|}{\|\vec{b}\|} \tag{6}$$

De totale fout van het gefilterde robotpad wordt bepaald door de som van alle afzonderlijke fouten. De gemiddelde fout is de totale fout gedeeld door het aantal gefilterde punten. De gemiddelde fout wordt gebruikt als indicatie voor de nauwkeurigheid van het gefilterde pad.

3.2 Vergelijking algoritmes

Aan de hand van zeven testen worden de verschillende algoritmes met elkaar vergeleken. Deze testen bestaan uit puntenbestanden die paden beschrijven met rechte en cirkelvormige bewegingen in een driedimensionale ruimte. Allereerst moeten de algoritmes geprogrammeerd worden in een C# bestand. Vervolgens wordt de foutfunctie geïmplementeerd. Elk algoritme filtert de verschillende bestanden met drie verschillende parameters. Hierbij wordt steeds de tijd bepaald in milliseconde, het aantal overgebleven punten en de gemiddelde fout in millimeter. Doordat het filteren van de punten op een aparte computer wordt uitgevoerd, is de tijd afhankelijk van de kracht van de CPU en het in beslag genomen werkgeheugen door andere programma's. Om de testen zo representatief mogelijk te maken, worden alle openstaande programma's op de PC afgesloten en worden de zeven testen meteen na elkaar uitgevoerd. Tabel 3 en Tabel 4 geven de resultaten weer.

Tabel 3: Resultaten van testen van alle algoritmes met verschillende parameters

	TEST 1			TEST 2			TEST 3			TEST 4		
	punten 76001	tijd (ms)	fout (mm)	punten 91001	tijd (ms)	fout (mm)	punten 26001	tijd (ms)	fout (mm)	punten 48001	tijd (ms)	fout (mm)
Douglas-Peucker	19476	106,32	0,061	24942	126,60	0,062	8376	60,87	0,064	16484	62,37	0,063
	8678	89,08	0,155	11406	78,82	0,155	3819	23,92	0,162	8673	40,43	0,160
	3871	93,79	0,350	5068	65,58	0,355	1722	24,26	0,360	3888	46,28	0,371
Radial-distance	22131	4,76	0,147	26885	3,99	0,154	7883	2,80	0,174	14963	2,99	0,187
	10392	1,39	0,181	12696	1,24	0,187	3788	0,47	0,217	6868	0,68	0,277
	4970	0,77	0,277	5951	0,96	0,293	1743	0,30	0,344	3488	0,50	0,382
Nth point	25335	0,92	0,198	30335	0,84	0,210	8669	0,70	0,245	16002	0,71	0,280
	10859	0,13	0,341	13002	0,13	0,359	3717	0,04	0,417	6859	0,07	0,420
	5069	0,09	0,494	6069	0,06	0,520	1736	0,02	0,581	3202	0,03	0,541
Visvalingam	15595	18,81	0,203	21075	21,10	0,198	7989	7,25	0,180	18307	19,09	0,205
	7200	17,68	0,348	9524	19,71	0,347	3383	5,78	0,340	11370	21,42	0,301
	3194	21,09	0,550	4200	23,67	0,555	1522	5,68	0,536	3998	22,02	0,543
Perpendicular distance	19596	34,27	0,096	25011	38,32	0,099	8602	9,83	0,105	16096	14,90	0,090
	8590	42,41	0,205	11159	35,20	0,210	3899	7,90	0,223	8785	18,20	0,236
	4200	25,64	0,354	5376	22,40	0,364	1690	6,83	0,408	3451	13,68	0,448
Ruemann-Witkam	16696	4,23	0,212	22188	4,12	0,208	7992	2,44	0,205	18376	3,22	0,190
	7275	2,70	0,372	9760	2,06	0,367	3660	0,93	0,366	10447	1,84	0,380
	2926	2,63	0,596	3928	3,17	0,594	1452	0,85	0,599	4368	1,65	0,614
Opheim	17385	7,44	0,307	22500	11,97	0,308	7857	5,12	0,304	18262	6,23	0,293
	8381	3,96	0,521	10817	5,71	0,530	3710	1,64	0,520	9379	3,28	0,515
	3018	5,19	0,585	4055	6,31	0,581	1501	1,86	0,581	4485	3,63	0,597
Lang	17488	16,42	0,071	22847	16,60	0,072	7989	9,53	0,073	17651	13,65	0,058
	7326	5,74	0,167	9392	7,61	0,171	3289	3,90	0,178	9846	8,65	0,141
	5114	3,14	0,431	6147	3,91	0,452	1766	1,32	0,496	3284	2,99	0,465

Tabel 4: Vervolg tabel 3

	TEST 5			TEST 6			TEST 7			Parameter
	punten 13001	tijd (ms)	fout (mm)	punten 18001	tijd (ms)	fout (mm)	punten 1000000	tijd (ms)	fout (mm)	
Douglas-Peucker	5847	32,08	0,073	7832	51,14	0,072	39618	573,19	0,048	0,22
	2826	8,11	0,175	3808	15,47	0,172	16067	468,14	0,134	0,63
	1254	7,59	0,398	1669	11,58	0,392	6495	404,07	0,342	1,5
Radial-distance	4029	2,80	0,256	5595	3,14	0,239	278198	16,27	0,022	1,42
	1881	0,32	0,331	2645	0,42	0,307	120772	10,59	0,033	2,3
	974	0,15	0,458	1337	0,35	0,445	54344	6,54	0,051	2,7
Nth point	4335	0,63	0,349	6002	0,64	0,332	333335	2,21	0,033	3
	1859	0,02	0,517	2574	0,04	0,527	142859	0,85	0,063	7
	869	0,01	0,709	1202	0,02	0,693	66669	0,39	0,112	15
Visvalingam	6496	4,27	0,140	8731	8,33	0,141	22126	233,34	0,297	0,09
	2900	3,03	0,319	3737	9,59	0,318	10808	324,32	0,447	0,2
	1167	3,65	0,559	1551	14,57	0,540	4676	586,07	0,596	0,5
Perpendicular distance	6052	6,90	0,118	8088	12,64	0,118	33381	237,41	0,066	0,24
	2887	3,68	0,265	3867	4,39	0,262	14091	259,46	0,153	0,65
	1177	3,05	0,480	1537	3,14	0,487	8605	328,55	0,227	1,42
Ruemann-Witkam	6362	5,71	0,190	8503	1,64	0,195	18121	21,55	0,240	0,18
	3053	0,57	0,350	4053	0,47	0,353	7949	24,91	0,397	0,4
	1245	0,50	0,583	1644	0,41	0,578	3265	19,94	0,614	0,95
Opheim	5933	4,17	0,287	7861	2,58	0,297	31592	48,45	0,336	0,27 en 2
	2869	2,15	0,516	3823	0,73	0,514	14615	30,28	0,546	0,68 en 2
	1272	2,98	0,574	1705	0,94	0,564	3374	44,40	0,605	0,92 en 4
Lang	6036	8,01	0,077	8034	4,91	0,077	74004	30,42	0,033	0,17 en 14
	2622	2,54	0,193	3410	2,26	0,193	72154	27,71	0,053	0,46 en 14
	892	0,73	0,608	1236	1,02	0,610	66731	26,95	0,098	1,9 en 14

Vervolgens wordt er voor elk algoritme met een bepaalde parameterinstelling de gemiddelde tijd, fout en aantal overgebleven punten bepaald. Tabel 5 geeft de gemiddelde waardes voor elk algoritme weer. De *simplification* is het gemiddelde van de verhouding van de overgebleven punten op het totaal aantal punten van alle testen. De simplification geeft dus weer hoeveel procent van de punten gemiddeld wordt overgehouden. De parameterinstellingen worden zo gekozen dat de simplification voor de verschillende algoritmes ongeveer gelijk is. Op deze manier kunnen ze met elkaar vergeleken worden. Het kan echter gebeuren dat twee algoritmes evenveel punten overhouden, maar dit wil niet zeggen dat ze dezelfde punten behouden. Hierdoor zal de gemiddelde fout ook verschillend zijn en kan er gezegd worden dat het ene algoritme nauwkeuriger is dan een ander. Uit Tabel 5 blijkt dat naarmate de parameters verhoogd worden, wat betekent dat de nauwkeurigheid van het pad daalt, de algoritmes sneller zijn. Het nauwkeurigste algoritme voor kleine toleranties is de Douglas-Peucker, echter is dit algoritme ook het traagst.

Tabel 5: Samenvattende resultaten van testen met alle algoritmes met verschillende parameters

	PARAMETER	GEM FOUT (mm)	GEM TIJD (ms)	SIMPLIFICATION (%)
Douglas-Peucker	0,22	0,063	144,65	30,3
	0,63	0,159	103,42	14,5
	1,5	0,367	93,3	6,4
Radial-distance	1,42	0,168	5,25	30
	2,3	0,219	2,16	14
	2,7	0,321	1,37	6,8
Nth point	3	0,235	0,95	33,3
	7	0,378	0,18	14,3
	15	0,521	0,09	6,7
Visvalingam	0,09	0,195	44,6	30,5
	0,2	0,346	57,36	14,4
	0,5	0,554	96,68	5,9
Perpendicular distance	0,24	0,099	50,61	30,7
	0,65	0,222	53,03	14,6
	1,42	0,396	57,61	6,2
Ruemann-Witkam	0,18	0,206	6,13	30,5
	0,4	0,369	4,78	14,7
	0,95	0,597	4,16	6
Opheim	0,27 en 2	0,305	12,28	29,8
	0,68 en 2	0,523	6,82	14,5
	0,92 en 4	0,584	9,33	6,2
Lang	0,17 en 14	0,066	14,22	30,6
	0,46 en 14	0,156	8,35	14,2
	1,9 en 14	0,451	5,72	6,8

3.3 Keuze algoritme

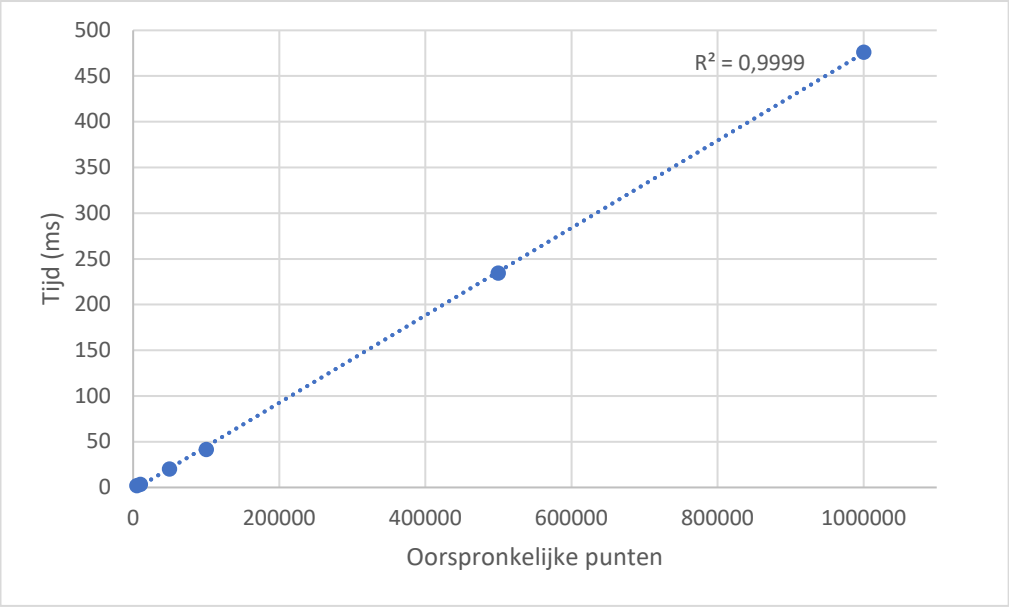
Zoals eerder aangehaald, wordt de keuze van het algoritme bepaald aan de hand van drie criteria. Tabel 6 geeft per algoritme een beoordeling van die criteria weer. De eenvoud van de parameter is van belang, omdat de parameter per bestand aangepast moet worden. Indien het puntenbestand veel rechte paden bevat, mag de parameter worden vergroot. Indien het veel cirkelvormige paden bevat, moet er een kleine waarde voor de parameters ingesteld worden. Voor de Douglas-Peucker, Radial-Distance, Perpendicular distance en Ruemann-Witkam zijn de parameters afstanden. Deze zijn eenvoudig te interpreteren. De parameter van Visvalingam is een oppervlakte, hierdoor is het bepalen van de parameter moeilijker. Voor Opheim en Lang moeten er twee parameters bepaald worden. Het belangrijkste criterium voor deze toepassing is de nauwkeurigheid. Merk op dat bij het berekenen van de fout wordt verondersteld dat de robot enkel lineaire bewegingen maakt. Hierdoor zijn de bekomen resultaten enkel van toepassing indien de robot in lineaire modus staat. Het optimale algoritme voor splines is moeilijk te bepalen, dit komt doordat er bij splines meerdere factoren invloed hebben op de baan van de robot. Ook is de exacte berekening die de robot gebruikt voor het berekenen van een spline-pad een geheim binnen KUKA. Hierdoor zal er voor de werking met splines onderzocht moeten worden welke invloed bepaalde punten op het pad hebben. De snelheid is niet van groot belang, omdat het puntenbestand maar één keer gefilterd moet worden. Ook blijkt uit test 7 dat het algoritme één miljoen punten kan filteren in ongeveer een halve seconde of minder, dit is nog steeds een snelle tijd. Uit de eerste resultaten weergegeven in Tabel 6 blijkt dat het Douglas-Peucker-algoritme het meest geschikte algoritme is voor deze toepassing.

Tabel 6: Vergelijking van alle algoritmes op basis van snelheid, nauwkeurigheid en eenvoud van de parameter

	Snelheid	**Nauwkeurigheid**	**Eenvoud parameter**
Douglas-Peucker	-	++	++
Radial-distance	+	+	++
Nth point	++	--	++
Visvalingam	+	-	+
Perpendicular distance	+	+	++
Ruemann-Witkam	+	-	++
Opheim	+	--	--
Lang	+	+	--

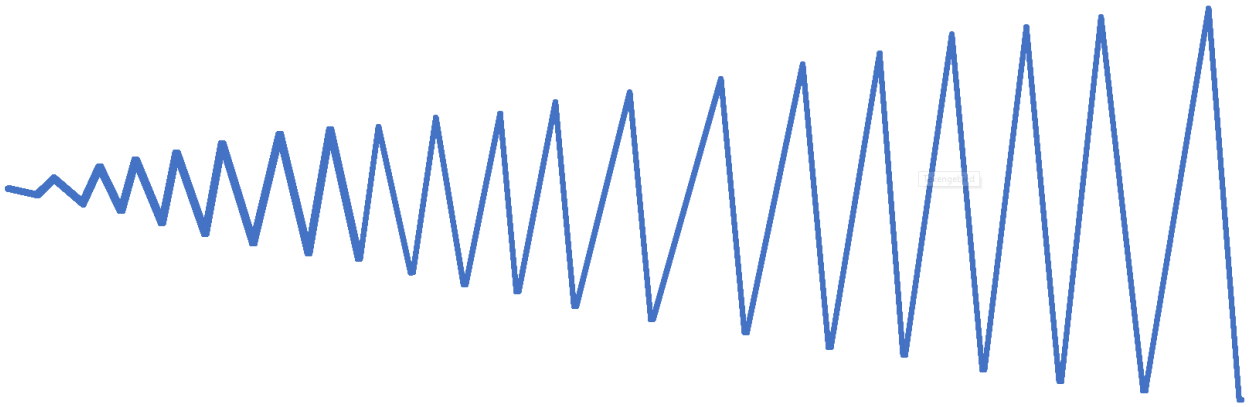
Om zekerheid te krijgen over de snelheid van het Douglas-Peucker-algoritme werden er bijkomende testen uitgevoerd. In totaal werden er twee verschillende testen uitgevoerd waarin telkens zes verschillende puntenbestanden gefilterd werden. Voor elk bestand werd zowel de tijd als het aantal oorspronkelijke punten bijgehouden.

Tijdens de eerste test werd er gebruik gemaakt van zes willekeurige puntenbestanden. Voor elk bestand werd de tijd uitgezet in functie van het aantal oorspronkelijke punten. Figuur 22 geeft deze grafiek weer. Hierin is een trendlijn getekend door de punten en wordt ook de R^2-waarde weergegeven. Deze waarde toont aan hoe goed de correlatie is tussen de trendlijn en de punten op de grafiek. Uit deze grafiek blijkt dat er een lineair verband bestaat tussen het aantal te filteren punten en de tijd. Hieruit volgt dat de complexiteit van het Douglas-Peucker-algoritme O(n) is voor deze willekeurige puntenbestanden.

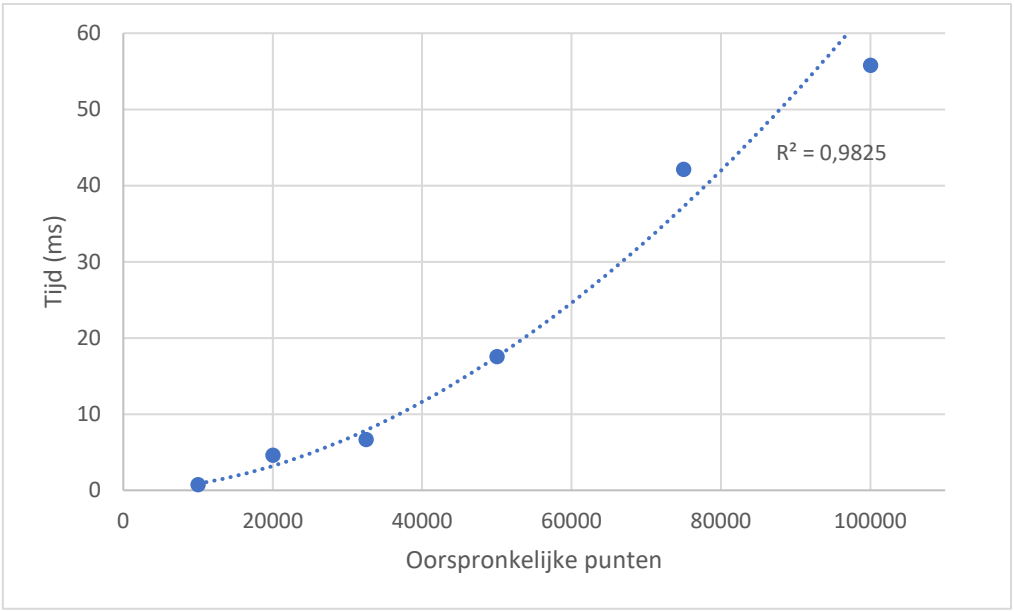


Figuur 22: Willekeurig scenario voor de complexiteit van Douglas-Peucker

Deze resultaten geven echter geen uitsluitsel over de complexiteit en de snelheid van het Douglas-Peucker-algoritme. Om ook het slechts mogelijke geval te kennen wordt een tweede test uitgevoerd. Om dit geval te simuleren wordt er gebruik gemaakt van zes puntenbestanden die een zigzagvorm maken zoals weergegeven in Figuur 23. Het eerste puntenbestand met het minst aantal punten begint met een kleine zigzag, bij elk volgend bestand zal deze zigzag meer uitbreiden. Doordat de zigzag telkens uitbreidt en er dus meer punten instaan, zal het algoritme telkens meer iteraties moeten uitvoeren om de punten te filteren. Opnieuw wordt er voor elk bestand de tijd uitgezet in functie van het aantal oorspronkelijke punten. Figuur 24 geeft het resultaat voor het slechts mogelijke scenario van Douglas-Peucker weer. Opnieuw is er een trendlijn getekend en wordt de R^2-waarde weergegeven. Hieruit volgt dat de tijd nu exponentieel stijgt ten opzichte van het aantal oorspronkelijke punten. De complexiteit van het Douglas-Peucker-algoritme is in het slechtste geval dus gelijk aan $O(n^2)$.



Figuur 23: Slechts mogelijke geval voor het Douglas-Peucker-algoritme



Figuur 24: Slechts mogelijke scenario voor de complexiteit van Douglas-Peucker

Op basis van deze testen kan geconcludeerd worden dat de complexiteit van het Douglas-Peucker-algoritme afhankelijk is van de ligging van de punten. Uit verdere testen blijkt ook dat de tijd niet alleen afhankelijk is van het aantal oorspronkelijke punten, maar ook van het aantal gefilterde punten. Volgens [17] blijkt dat de complexiteit van het Douglas-Peucker-algoritme gemiddeld O(n log m) is, waarbij n het aantal oorspronkelijke punten is en m het aantal punten na filteren. In het slechtste geval is de complexiteit gelijk aan O(nm). Deze slechtste complexiteit komt enkel voor in uitzonderlijke gevallen zoals een zaagtandvorm. Deze vormen komen amper of zelfs niet voor in freesprocessen. Hierdoor kan er geconcludeerd worden dat het Douglas-Peucker-algoritme een goed algoritme is voor het filteren van punten in een freesproces.

4 Code

4.1 Punten inlezen

Zoals eerder vermeld bevinden de punten zich in een CSV-bestand. Om de punten te filteren moeten ze eerst worden ingelezen in het programma. In het CSV-bestand worden de verschillende variabelen gescheiden door middel van een puntkomma. Een punt heeft in totaal 21 verschillende eigenschappen:

- nieuw volgnummer,
- X, Y, Z coördinaten,
- a, b, c: verdraaiingen rond de assen,
- oorspronkelijke volgnummer,
- voedingssnelheid,
- spindel aan of uit,
- draaisnelheid van de spindel,
- draairichting van de spindel,
- koeling,
- tool nummer,
- versnelling,
- E1, E2, E3, E4, E5, E6: waardes van de externe assen.

Eerder werd vermeld dat er 26 verschillende parameters zijn. In deze masterproef is er enkel gewerkt met bovenstaande 21 parameters. Hierdoor zal het programma enkel werken als het CSV-bestand deze 21 parameters bevat. Ook moeten deze parameters in het CSV-bestand in de juiste volgorde staan. De juiste volgorde is zoals hierboven vermeld. Om berekeningen op deze punten uit te voeren, wordt er een klasse Points gecreëerd. Deze klasse heeft bovenstaande eigenschappen als variabelen. Door middel van de geïmplementeerde library StreamReader kunnen CSV-bestanden ingelezen worden. Vervolgens worden alle 21 eigenschappen ingelezen en in variabelen geplaatst. Deze variabelen worden gebruikt om een object van de klasse “Points” te maken. Door een *array* te definiëren, die enkel objecten van de klasse Points mag bevatten, kunnen alle punten worden opgeslagen in het programma.

De volledige code die gebruikt wordt voor het inlezen van de punten staat in Bijlage *A*.

4.2 Implementatie Douglas-Peucker-algoritme

Uit de testen bleek dat het Douglas-Peucker-algoritme het beste algoritme is om punten te filteren uit een robotpad. Vervolgens wordt dit algoritme geïmplementeerd in het programma. Het algoritme werkt op basis van de loodrechte afstand van een punt tot een rechte. Echter werd deze functie al geïmplementeerd bij de testen voor de keuze van het algoritme in hoofdstuk 3.1. Aan het algoritme wordt één variabele meegegeven namelijk, de tolerantie waarmee de punten gefilterd moeten worden. De tolerantie kan door middel van een XML-bestand ingegeven en aangepast worden. Deze tolerantie wordt vervolgens ingelezen in het programma.

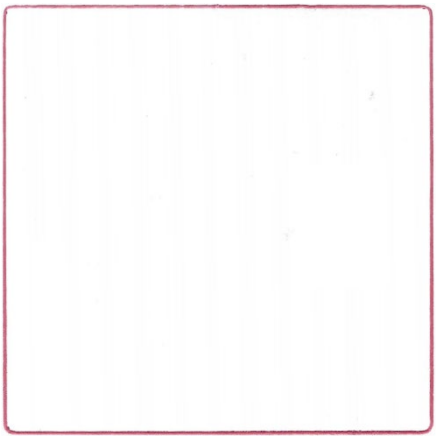
Eerst wordt er een bitarray aangemaakt, die dezelfde grote heeft als de array waarin zich de oorspronkelijke punten bevinden. Elke bit in de bitarray krijgt standaard een waarde *false*. Aangezien bij Douglas-Peucker het eerste en laatste punt altijd bijgehouden worden, wordt de waarde van de eerste en laatste bit van de bitarray aangepast naar *true*. Hierna wordt er een *stack* aangemaakt waarin zich alleen maar objecten van een klasse Range kunnen bevinden. Een stack is ongeveer hetzelfde als een list, echter zijn er belangrijke verschillen. Een stack heeft een *push* en *pop* methode. Door de push methode wordt er een object op de eerste plaats in de stack geplaatst. Het object dat zich eerst op de eerste plaats bevond, wordt verplaatst naar de tweede plaats enzovoort. De pop methode verwijdert het

object dat op de eerste plaats staat in de stack. De klasse Range heeft twee variabelen namelijk, een beginwaarde en een eindwaarde. Het eerste element dat in de stack wordt geplaatst is een range met een beginwaarde gelijk aan nul en een eindwaarde gelijk aan het totaal aantal punten min één. Deze range wordt meteen uit de stack verwijderd en in een variabele geplaatst. Vervolgens wordt van alle punten de loodrechte afstand tot de rechte, die door het eerste punt en het laatste punt gaat, bepaald. Hiervan wordt de langste afstand opgeslagen in een variabele. Indien deze afstand groter is dan de tolerantie, moet dit punt behouden worden. Dit gebeurt door op deze plaats in de bitarray de waarde aan te passen naar true. Vervolgens worden er twee nieuwe range objecten in de stack geplaatst, één met een range van het beginpunt tot het punt dat behouden moest worden en één range van het punt dat behouden moest worden tot het eindpunt. Hierna wordt de eerste range weer uit de stack gehaald en wordt de maximale afstand van elk punt in die range bepaald. Het punt met de maximale afstand wordt weer behouden en vervolgens worden er weer twee nieuwe range objecten aangemaakt en in de stack geplaatst. Dit proces blijft doorgaan totdat er geen objecten meer in de stack zitten. Indien het proces beëindigd is, wordt de bitarray terug gegeven. Elk punt dat op de overeenkomende *index* in de bitarray de waarde true heeft moet behouden blijven. De punten die op de overeenkomende index in de bitarray een false heeft, moeten verwijderd worden volgens Douglas-Peucker.

De volledige code voor het filteren volgens het Douglas-Peucker-algoritme staat in Bijlage *B*.

4.3 Implementatie hoekalgoritme

Door het gebruik van KUKA.CAMrob worden de paden van de robot automatisch in lineaire beweging met luswerking berekend. Omdat de robot met luswerking werkt, zal de eindeffector niet door alle punten gaan. Hierdoor krijgt men ongewenste afrondingen in het pad, echter zal de robot zijn pad sneller kunnen uitvoeren. Figuur 25 geeft een voorbeeld van de afrondingen weer.



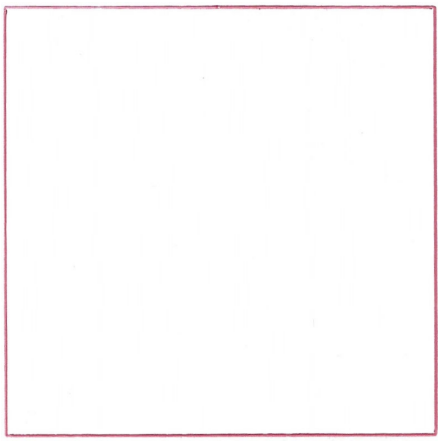
Figuur 25: Vierkant zonder punten voor en na het hoekpunt

De implementatie van het hoekalgoritme zal bovenstaand probleem oplossen. Het afronden van de hoeken wordt tegengegaan door de punten, die zich voor en na de hoek bevinden, te behouden. Het programma gaat voor elk punt de hoek berekenen, met behulp van volgende formule:

$$cos(\alpha) = \frac{\vec{V} \cdot \vec{W}}{\|\vec{V}\| \cdot \|\vec{W}\|} \tag{7}$$

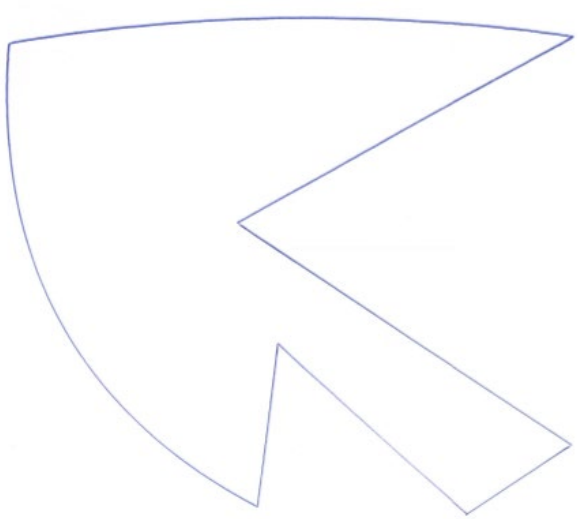
Hierbij zijn *V* en W vectoren en α de hoek tussen de twee vectoren. De vector V is de vector gaande van punt n tot punt n+1. Vector W is de vector gaande van punt n+1 tot punt n+2. Indien de hoek kleiner is dan een opgegeven waarde, behoudt het algoritme de punten voor en na het punt. Ook deze waarde

wordt als parameter meegegeven in een XML-bestand. Figuur 26 geeft een voorbeeld weer waarbij het hoekalgoritme is geïmplementeerd.



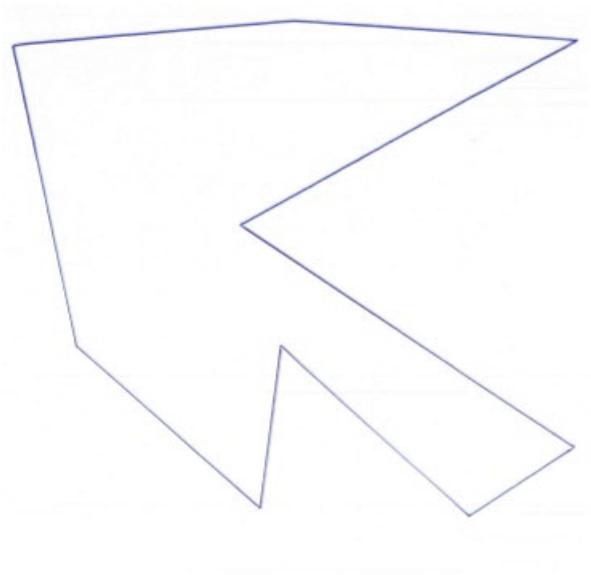
Figuur 26: Vierkant met punten voor en na het hoekpunt

Echter is bovenstaand algoritme geen correcte manier om de afrondingen te onderdrukken. Indien de hoek tussen drie punten groter is dan de parameter, zal het algoritme de punten voor en na het punt niet behouden. Hierdoor zal de robot deze hoek afronden. Figuur 27 geeft een voorbeeld weer waarbij de parameter van het hoekalgoritme gelijk is aan 140°. De hoeken die groter zijn dan 140° zullen afgerond worden getekend.



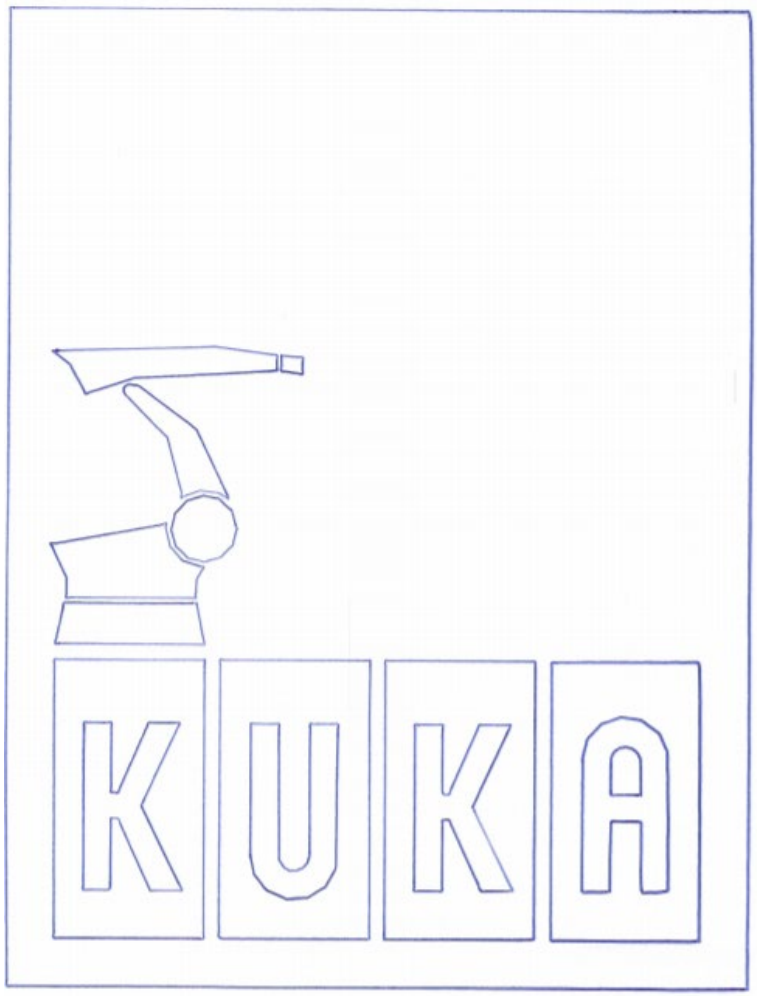
Figuur 27: Verschillende hoeken gefilterd: DPT = 0,1 mm, hoektolerantie = 140°, oriëntatietolerantie = 140°

Bovenstaand probleem kan opgelost worden door de parameter van het hoekalgoritme te verhogen naar 179°. Het resultaat wordt weer gegeven in Figuur 28.



Figuur 28: Verschillende hoeken gefilterd: DPT = 0,1 mm, hoektolerantie = 179°, oriëntatietolerantie = 140°

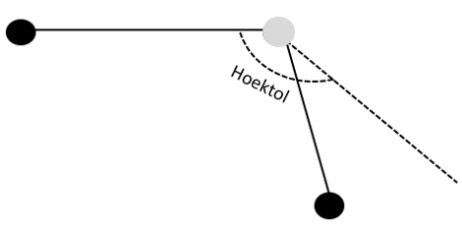
Het verhogen van de parameter is echter een slecht idee, omdat het algoritme een kromme beweging beschouwt als een set van hoeken. Het algoritme zal voor en na elk punt in de kromme een punt plaatsen, waardoor de kromme hoekig wordt getekend. Figuur 29 geeft een voorbeeld waarbij de parameter op 179° wordt ingesteld. Hier is duidelijk waar te nemen dat de kromme lijnen als rechte lijnen worden getekend.



Figuur 29: KUKA logo gefilterd: DPT = 0,1 mm, hoektolerantie = 179°, oriëntatietolerantie = 140°

Om het probleem op te lossen moet het algoritme kunnen detecteren of het punt tot een hoek of kromme behoort. Om te bepalen of het punt een hoekpunt is of deel uitmaakt van een cirkelvormige beweging, worden drie verschillende hoeken bepaald. Allereerst wordt de hoek van het punt zelf berekend. Ten tweede wordt de hoek van het vorige punt berekend en tot slot wordt de hoek van het volgende punt berekend. Net zoals bij de vorige algoritmes wordt er via het XML-bestand een parameter meegegeven. Er zijn echter drie situaties mogelijk.

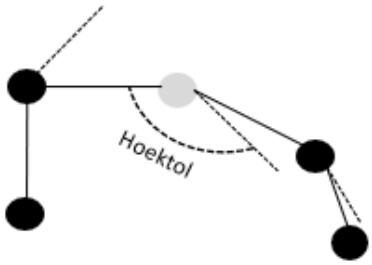
- De hoek van het punt is kleiner dan de parameter. Dit betekent dat de hoek een hoekpunt is en vervolgens worden de punten voor en na het punt behouden. Figuur 30 visualiseert een voorbeeld waarbij de hoek kleiner is dan de tolerantie.



Figuur 30: Hoekalgoritme situatie 1: hoek kleiner dan hoektolerantie

- De hoek is groter dan de parameter, de hoek van het vorige punt is kleiner dan de parameter en de hoek van het volgend punt is ook kleiner dan de parameter. Hieruit volgt dat het punt ook een hoekpunt is. Hierdoor moeten de punten voor en na het punt behouden worden. Figuur 31

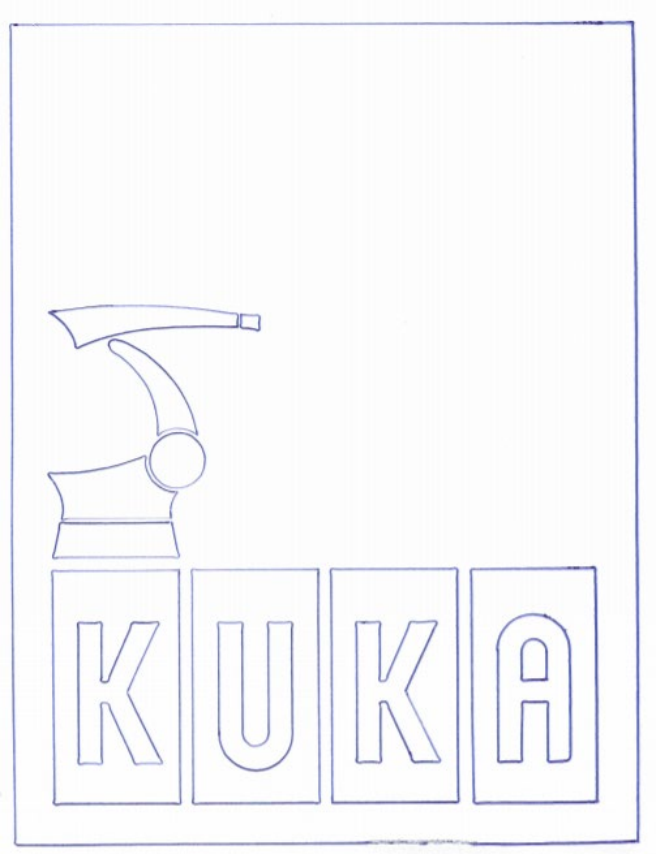
visualiseert een voorbeeld waarbij de hoek groter is dan de tolerantie en de vorige en volgende hoek kleiner zijn dan de tolerantie.



Figuur 31: Hoekalgoritme situatie 2: hoek groter dan hoektolerantie en vorige en volgende hoek kleiner dan hoektolerantie

- In alle andere situaties moet er niets gebeuren. Hier zal het punt namelijk deel uitmaken van een cirkelvormige beweging.

De parameter van het hoekalgoritme ligt rond de 140 graden, deze parameter is afhankelijk van de tolerantie van de Douglas-Peucker. Indien de tolerantie groot is, zullen er zich minder punten in het cirkelvormig pad bevinden en zal de hoek tussen de punten kleiner zijn. Hierdoor zal de parameter verlaagd moeten worden. Figuur 32 geeft een voorbeeld weer waarbij het hoekalgoritme wordt geïmplementeerd.



Figuur 32: KUKA logo gefilterd: DPT = 0,1 mm, hoektolerantie = 140°, oriëntatietolerantie = 140°

De volledige code voor het filteren volgens het hoekalgoritme is terug te vinden in Bijlage *C*.

4.4 Implementatie oriëntatiealgoritme

Voorgaande filteralgoritmes houden enkel rekening met de XYZ-coördinaten van de punten. Echter kan de oriëntatie van de eindeffector ook wijzigen. Hiervoor wordt er een derde algoritme geïmplementeerd dat rekening houdt met de oriëntatieverandering van de eindeffector. Indien de oriëntatie verandering groter is dan een opgegeven tolerantie, die zich in het XML-bestand bevindt, moet dit punt behouden worden. Ook hier gebeurt dit door een true te schrijven op een bepaalde plaats in een bitarray.

De volledige code voor het filteren volgens het oriëntatiealgoritme staat in Bijlage *D*.

4.5 Implementatie spindelalgoritme

Zoals eerder vermeld, bevat een punt 21 verschillende variabelen. Eén van deze variabelen geeft aan of de spindel in dat punt moet draaien, een andere variabele bepaalt de draaisnelheid van de spindel en een laatste variabele bepaalt de draairichting. Indien in een punt één van deze parameters verandert, moet het programma dat punt behouden.

De verschillende algoritmes geven allen een bitarray terug van punten die behouden moeten worden. Het kan echter zijn dat het Douglas-Peucker-algoritme een bepaald punt verwijdert, maar dit punt juist belangrijk is voor het hoekalgoritme zodat de hoeken niet worden afgerond. Hierdoor wordt er een logische disjunctie uitgevoerd op de verschillende bitarrays. Op deze manier zal elk belangrijk punt behouden blijven.

De volledige code voor het filteren volgens het spindelalgoritme is terug te vinden in Bijlage *E*.

4.6 Tijdsbepaling

Om een idee te krijgen van de tijdswinst, verkregen door het filteren van de punten voor het freesproces uit te voeren, is een tijdsbepaling geïmplementeerd in de code. De robot heeft een maximale snelheid van 2 m/s, deze snelheid kan echter in de process file gelimiteerd worden door de voedingssnelheid aan te passen. Ook de maximale versnelling van de robot, die standaard 2,3 m/s² is, kan in de process file gelimiteerd worden. Om de tijd van het proces te berekenen moet als eerste de afstand tussen alle opeenvolgende punten berekend worden. Dit gebeurt aan de hand van volgende formule:

$$d = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2 + (z_1 - z_0)^2} \tag{8}$$

Hierna berekent het programma op welke afstand de robot zijn maximale snelheid behaalt. Het berekenen van deze afstand gebeurt met de volgende formule:

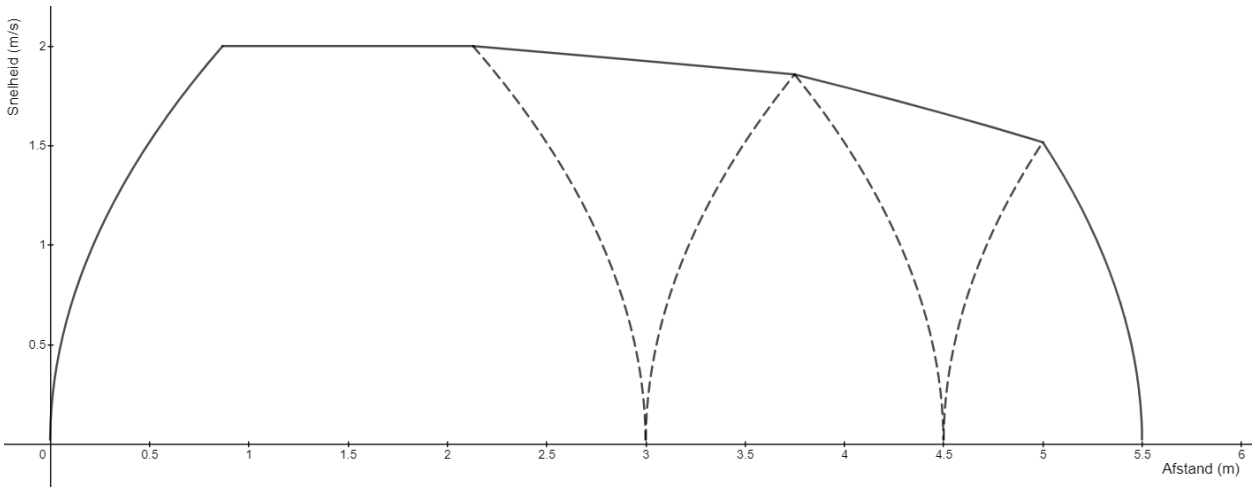
$$d_v = \frac{v_{max}}{2 \cdot a_{max}} \tag{9}$$

Indien deze afstand d_v kleiner is dan de afstand d gedeeld door 2, dan zal de robot in dit geval zijn maximale snelheid v_{max} behalen tussen de twee opeenvolgende punten. Deze maximale snelheid zal behaald worden op afstand d_v van het eerste punt. De robot zal terug beginnen met vertragen na een afstand $d - d_v$. Deze twee afstanden moeten allebei bijgehouden worden, samen met de maximale snelheid. Dit gebeurt in een array. Indien de afstand d_v echter groter is dan de afstand d gedeeld door 2, dan zal de robot zijn maximale snelheid v_{max} niet halen tussen de twee opeenvolgende punten. In dit geval kan de maximale snelheid die behaald kan worden als volgt berekend worden:

$$v = \sqrt{2 \cdot a_{max} \cdot \frac{d}{2}} \tag{10}$$

De afstand waar deze snelheid behaald wordt, is dan gelijk aan de afstand tussen de 2 punten gedeeld door 2. Deze afstand wordt dan samen met de snelheid die behaald wordt bijgehouden in de array.

Figuur 33 geeft een snelheid in functie van afstand grafiek weer van 3 punten op x = 3, x = 4,5 en x = 5,5. De stippellijnen zijn de tot hier toe berekende waardes voor de maximale snelheid en de afstand waarop deze snelheid bereikt wordt. De robot zal echter niet terug tot stilstand komen in elk punt. Doordat de robot in luswerking staat zal de robot zijn snelheid zo hoog mogelijk proberen te houden. De robot zal hierdoor met een constante snelheid vertragen of versnellen tussen twee opeenvolgende punten om de hierboven maximale snelheid te bekomen. De werkelijke snelheid van de robot wordt in Figuur 33 weergegeven door de volle lijn.



Figuur 33: Snelheid in functie van de afstand

De versnelling van de robot tussen twee opeenvolgende punten wordt als volgt berekend:

$$a_i = \frac{v_i^2 - v_{i-1}^2}{2 \cdot d} \tag{11}$$

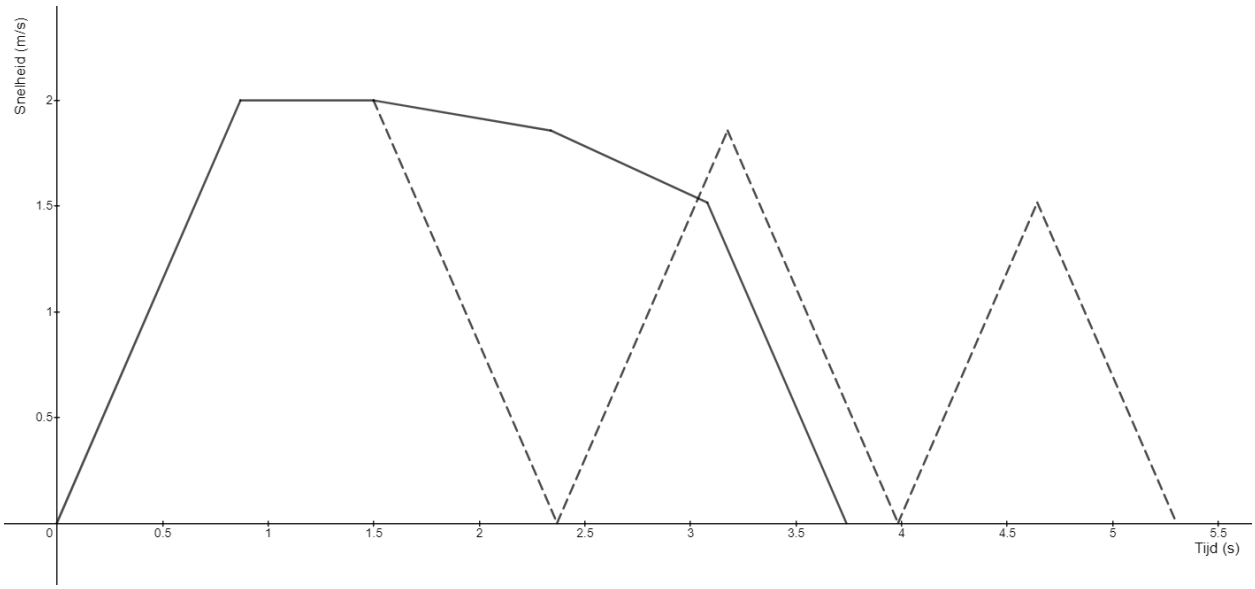
Ook deze versnelling wordt voor elk punt bepaald en bijgehouden in een array. Aan de hand van alle waardes in de arrays kan de tijd bepaald worden die de robot nodig heeft om tussen twee punten te bewegen. Er zijn hierbij 2 mogelijke gevallen: de snelheid kan constant blijven of de versnelling blijft constant. Indien de snelheid constant blijft kan de tijd voor dit deel als volgt berekend worden:

$$t = \frac{\Delta d}{v} \tag{12}$$

Indien de versnelling constant is, kan de tijd als volgt berekend worden:

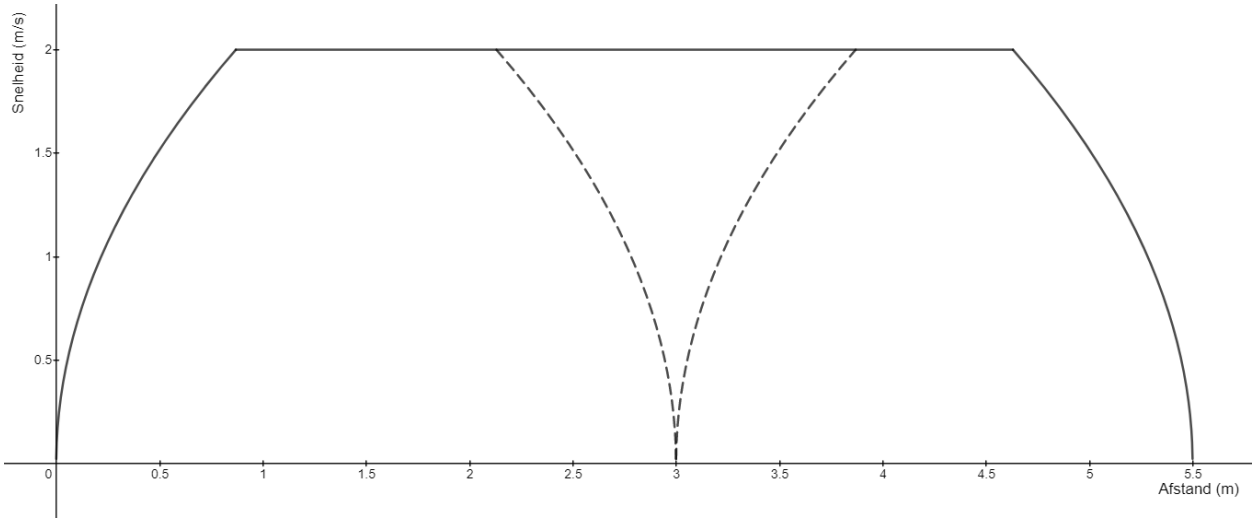
$$t = \frac{\Delta v}{a} \tag{13}$$

Deze tijd wordt voor elk punt bepaald en opgeteld, hierdoor bekomt men de totale tijd die nodig is om het proces af te werken. Figuur 34 geeft een grafiek van de snelheid in functie van de tijd. Opnieuw zijn de stippellijnen de snelheden die behaald worden indien de robot in elk punt tot stilstand komt. De volle lijnen geven de werkelijke tijd met luswerking weer indien de robot zijn snelheid zo hoog en constant mogelijk probeert te houden.



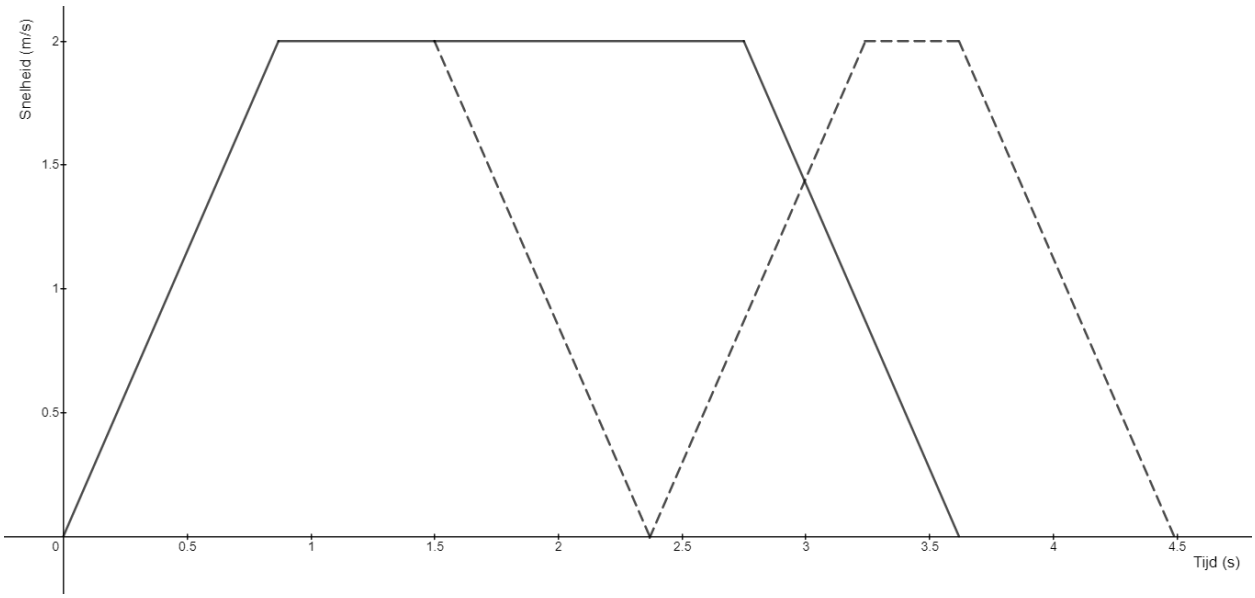
Figuur 34: Snelheid in functie van de tijd

Indien we nu het puntenbestand filteren waardoor punt 2 op x = 4,5 verwijderd wordt, dan toont Figuur 35 dat de robot nu ook zijn maximale snelheid bereikt tussen punt 1 en punt 3. De robot moet dus enkel versnellen op het begin en vertragen op het einde.



Figuur 35: Snelheid in functie van de afstand na filteren

Figuur 36 geeft de snelheid weer in functie van de tijd van de gefilterde punten. De robot zal zoals hierboven gezegd enkel op het begin versnellen en op het einde vertragen. Hiertussen blijft de snelheid constant. Dit geeft een kleine tijdswinst in vergelijking met de originele punten. Deze tijdswinst zal groter worden indien er meer punten gefilterd worden.



Figuur 36: Snelheid in functie van de tijd na filteren

Deze tijd is echter slechts een schatting en niet nauwkeurig. De maximale versnelling en snelheid van de robot is verder gelimiteerd door onder andere asversnellingen en het gewicht van de last. Hierdoor is het beter de procentuele tijdswinst te berekenen en te gebruiken om een idee te krijgen van de tijd die uitgespaard wordt door het filteren.

De code voor het bepalen van de tijd die de robot erover doet om het volledige pad uit te voeren is te vinden in Bijlage *F*.

4.7 Punten uitschrijven

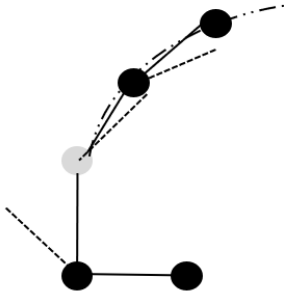
Na het filteren van de punten, moet de lijst van gefilterde punten opnieuw in een CSV-bestand geschreven worden. Hiervoor zal de lijst met punten in volgorde doorlopen worden, elke variabele die het punt bezit wordt gescheiden door een komma door middel van de Format functie. Het volgende punt zal telkens op een nieuwe lijn in het CSV-bestand beginnen.

De code voor het uitschrijven van de punten naar een CSV-bestand is terug te vinden in Bijlage *G*.

4.8 Aanpassing hoekalgoritme voor spline-bewegingen

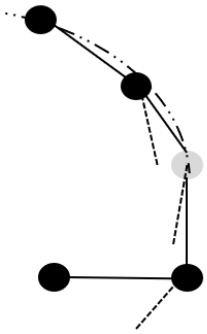
Indien er met splines gewerkt wordt, moet het hoekalgoritme aangepast worden. Bij het originele algoritme dat uitgelegd is in hoofdstuk 4.3 werden er punten voor en na het hoekpunt behouden. Voor splines moet er nog steeds een punt voor en na het hoekpunt staan. Echter zal bij spline de robot stoppen in een punt indien twee opeenvolgende punten dichter dan 0,1 mm van elkaar liggen. Hierdoor kan er geen gebruik gemaakt worden van originele punten en zal het programma zelf punten toevoegen, die op een afstand van 1 mm van het hoekpunt liggen. Bij splines zijn er namelijk extra situaties waarbij er punten toegevoegd moeten worden.

- De hoek is groter dan de parameter, de hoek van het vorige punt is kleiner dan de parameter en de hoek van het volgend punt is groter dan de tolerantie. Dit betekent dat het punt het beginpunt is van een cirkelvormig pad. Hierdoor moet het algoritme een punt ervoor toevoegen. Figuur 37 visualiseert een voorbeeld waarbij de hoek en volgende hoek groter zijn dan de tolerantie en de vorige hoek kleiner is dan de tolerantie.



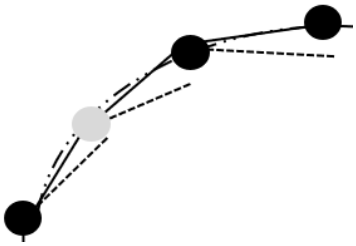
Figuur 37: Hoekalgoritme situatie 4: hoek en volgende hoek groter dan hoektolerantie en vorige hoek kleiner dan hoektolerantie

- De hoek is groter dan de parameter, de hoek van het vorige punt is groter dan de parameter en de hoek van het volgend punt is kleiner dan de parameter. Dit betekent dat het punt het eindpunt is van een cirkelvormig pad. Hierdoor moet het algoritme een punt erna toevoegen. Figuur 38 visualiseert een voorbeeld waarbij de hoek en vorige hoek groter zijn dan de tolerantie en de volgende hoek kleiner is dan de tolerantie.



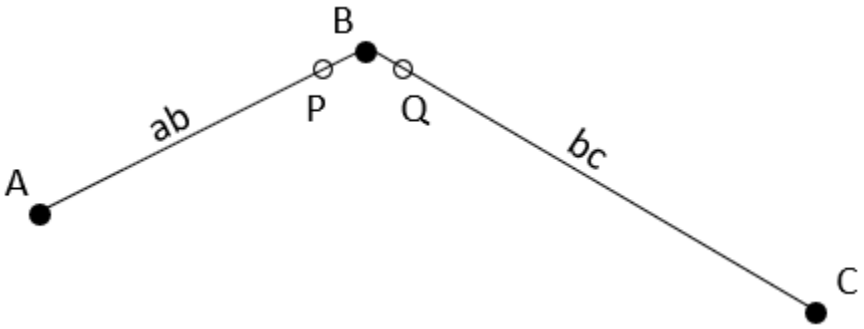
Figuur 38: Hoekalgoritme situatie 5: hoek en vorige hoek groter dan hoektolerantie en volgende hoek kleiner dan hoektolerantie

- De hoek is groter dan de parameter, de hoek van het vorige punt is groter dan de parameter en de hoek van het volgend punt is groter dan de parameter. Dit betekent dat het punt zich in een cirkelvormig pad bevindt, als volgt moet het algoritme niets doen. Figuur 39 visualiseert een voorbeeld waarbij de hoek, vorige hoek en volgende hoek groter zijn dan de tolerantie.



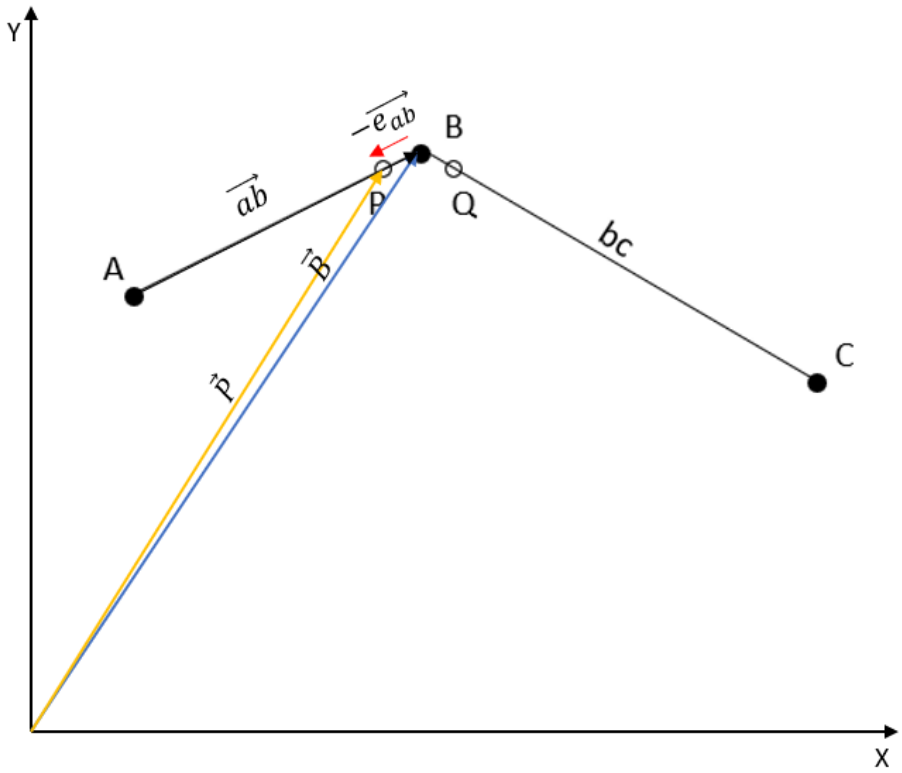
Figuur 39: Hoekalgoritme situatie 6: hoek, vorige hoek en volgende hoek groter dan hoektolerantie

Nadat er bepaald is waar het punt zich precies bevindt, worden er al dan niet voor en/of na dat punt punten toegevoegd. Figuur 40 geeft een voorbeeld weer waarbij punt B een hoekpunt is. Voor en na het punt B moeten dus punten toegevoegd worden. De punten die worden toegevoegd, liggen op de rechte ab en bc en op 1 mm van het punt B. Indien de afstand van het punt A tot het punt B kleiner is dan 2 mm, wordt het punt P niet toegevoegd. Hetzelfde geldt voor punt Q indien de afstand van punt B tot punt C kleiner is dan 2 mm.



Figuur 40: Aanpassing hoekalgoritme voor spline

Om de x-, y- en z-coördinaten van het punt P te bepalen wordt er gebruik gemaakt van vectoren. Eerst wordt de vector *ab* bepaald door gebruik te maken van de Vector3 library. Vervolgens wordt de eenheidsvector van vector *ab* bepaalt door de vector te delen door zijn lengte. De eenheidsvector van *ab* is een vector met dezelfde zin en richting, maar heeft een grootte van 1. Hierna wordt de vector *B* bepaalt, deze vector heeft als beginpunt de oorsprong en als eindpunt het punt *B*. Door de eenheidsvector van *ab* af te treken van de vector *B,* wordt de vector *P* bekomen. De coördinaten van het punt *P* zijn gelijk aan het eindpunt van de vector *P*. Figuur 41 geeft een voorbeeld weer voor het bepalen van een punt gelegen op een afstand van 1 mm van een ander punt.



Figuur 41: Meetkundige bepaling van een punt op 1 mm van een ander punt

Het zelfde principe wordt gebruikt om de coördinaten van het punt erna te bepalen. Echter wordt hierbij de eenheidsvector niet van de andere vector afgetrokken, maar erbij opgeteld. De volledige code voor de aanpassingen van het hoekalgoritme voor spline-bewegingen is terug te vinden in Bijlage *H*.

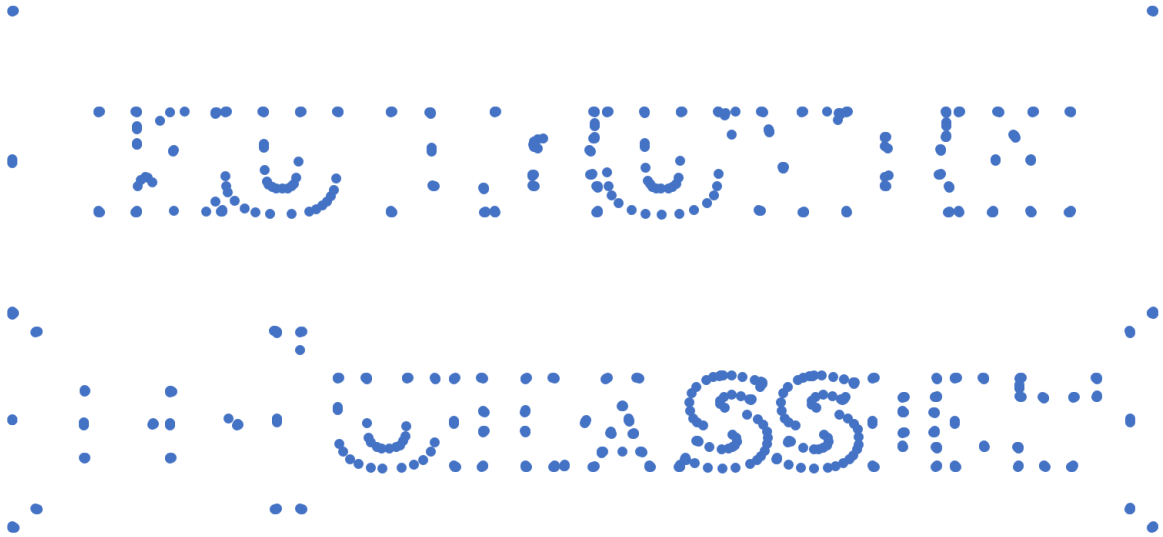
5 Lineaire beweging

5.1 2D-testen

Als eerste wordt het algoritme in 2D getest, hiervoor wordt een pen aan de robot bevestigd. Door middel van een online programma PathToPoints kunnen SVG bestanden omgezet worden in punten met coördinaten. In het programma kan meegegeven worden om de hoeveel millimeter een punt geplaatst moet worden. Figuur 42 geeft de in totaal 11804 punten in het KU Leuven en UHasselt logo weer.

Figuur 42: Punten KU Leuven en UHasselt logo

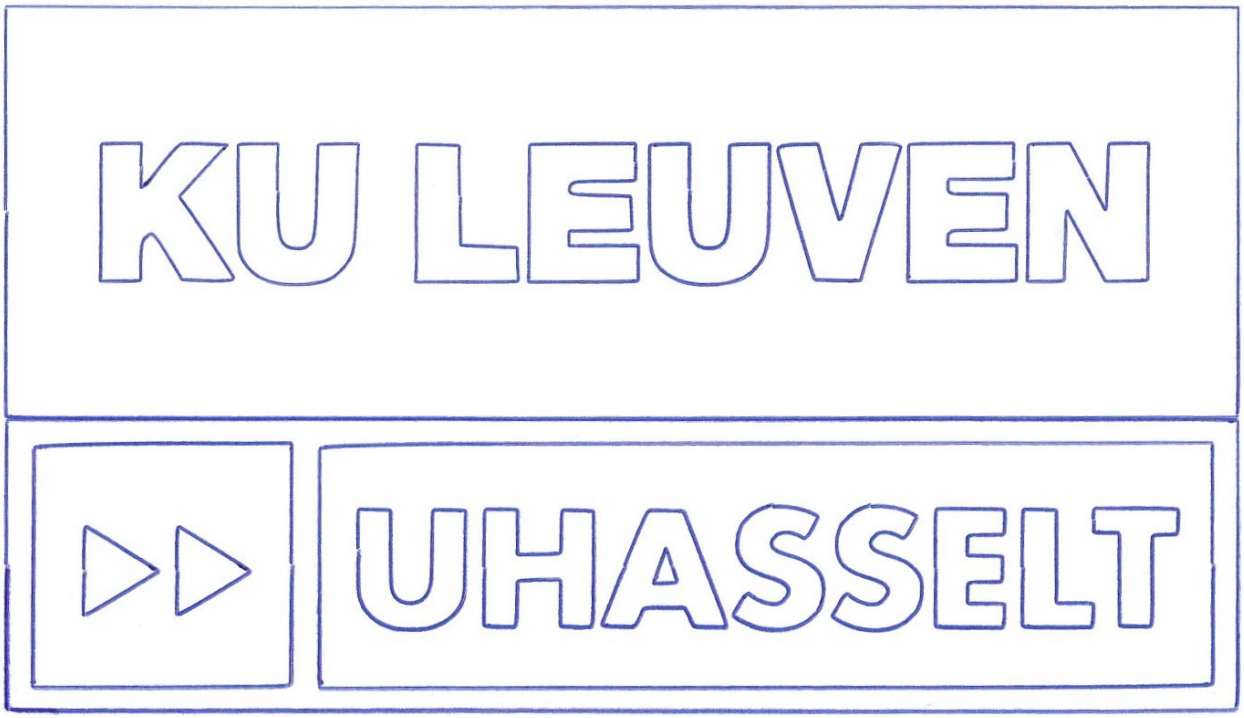
Vervolgens worden bovenstaande punten gefilterd door het filterprogramma. Figuur 43 geeft de gefilterde punten weer, hierbij is duidelijk op te merken dat het algoritme punten die op een rechte lijn liggen verwijdert en enkel de hoekpunten overhoudt. In elk hoekpunt worden bovendien drie punten bijgehouden door het hoekalgoritme, dit is op de figuur minder goed zichtbaar. Op plaatsen waar het pad een kromme bevat, worden veel punten behouden, dit is goed zichtbaar in de S'en en U's. Echter valt op dat bij de U's van het KU Leuven logo niet dezelfde punten worden bijgehouden. Dit komt doordat het online programma, dat de omzetting uitvoert, om de x millimeter een punt plaatst en het logo één SVG bestand is. Hierdoor staan in beide U's de oorspronkelijke punten op verschillende relatieve plaatsen.



Figuur 43: Gefilterde punten KU Leuven en UHasselt logo

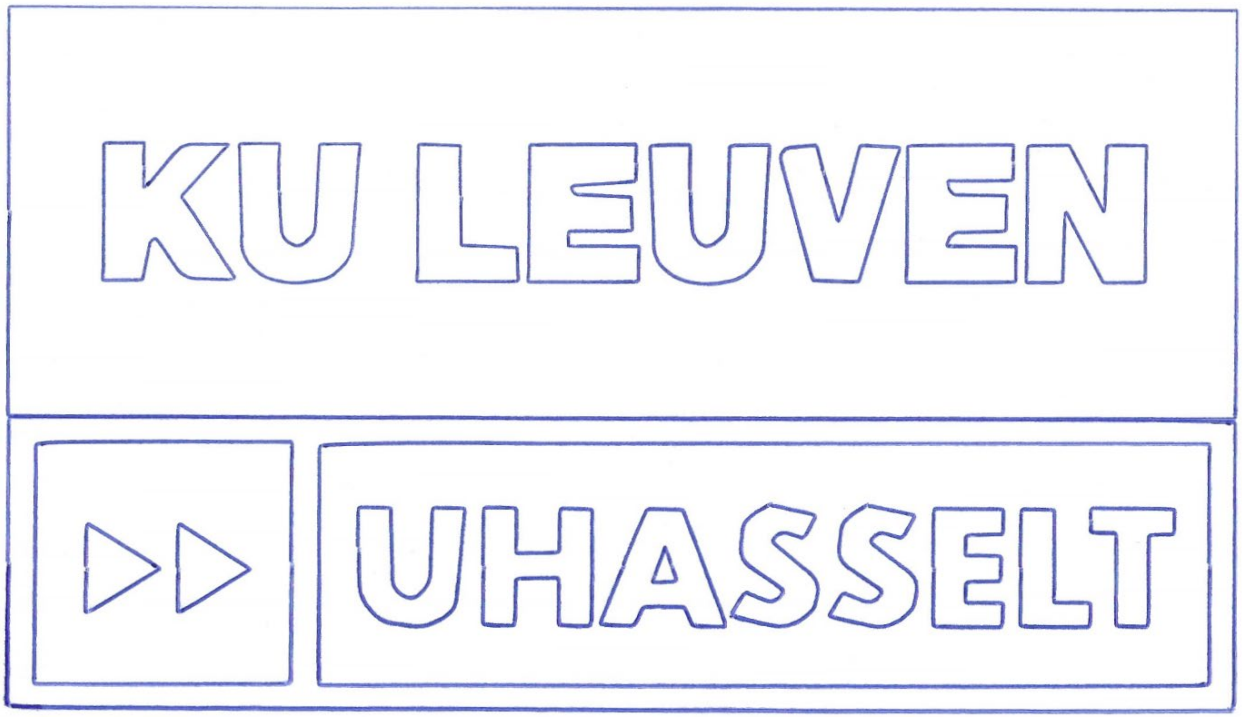
Om een goede vergelijking te bekomen tussen de verschillende testen, worden bepaalde parameters voor de lineaire testen gelijk gehouden. Zo is de maximale snelheid van de eindeffector 2 m/s, de robot werkt verder in automatische snelheid met een gereduceerde snelheid tot 10% van de maximale snelheid. Hierdoor berekent de robot wel het pad dat gevolgd zou worden indien de snelheid niet gereduceerd werd, maar wordt dit pad trager uitgevoerd zodat de pen niet breekt bij hoge snelheden.

Allereerst wordt het algoritme getest indien de robot in lineaire benaderende modus staat. Dit wil zeggen dat de robot tussen opeenvolgende punten een rechte gaat tekenen. De robot zal echter niet door elk punt gaan, maar de hoeken afronden indien er een hoek gemaakt wordt. Om een goede visuele nauwkeurigheidsverandering te kunnen waarnemen tussen de originele en de gefilterde punten worden eerst de ongefilterde punten getekend. Ook de verandering in tijd die de robot doet over het tekenen kan op deze manier eenvoudig vergeleken worden. Figuur 44 geeft het resultaat van de getekende, ongefilterde punten weer. De robot deed er 1 uur 43 minuten en 15 seconden over om beide logo's te tekenen.



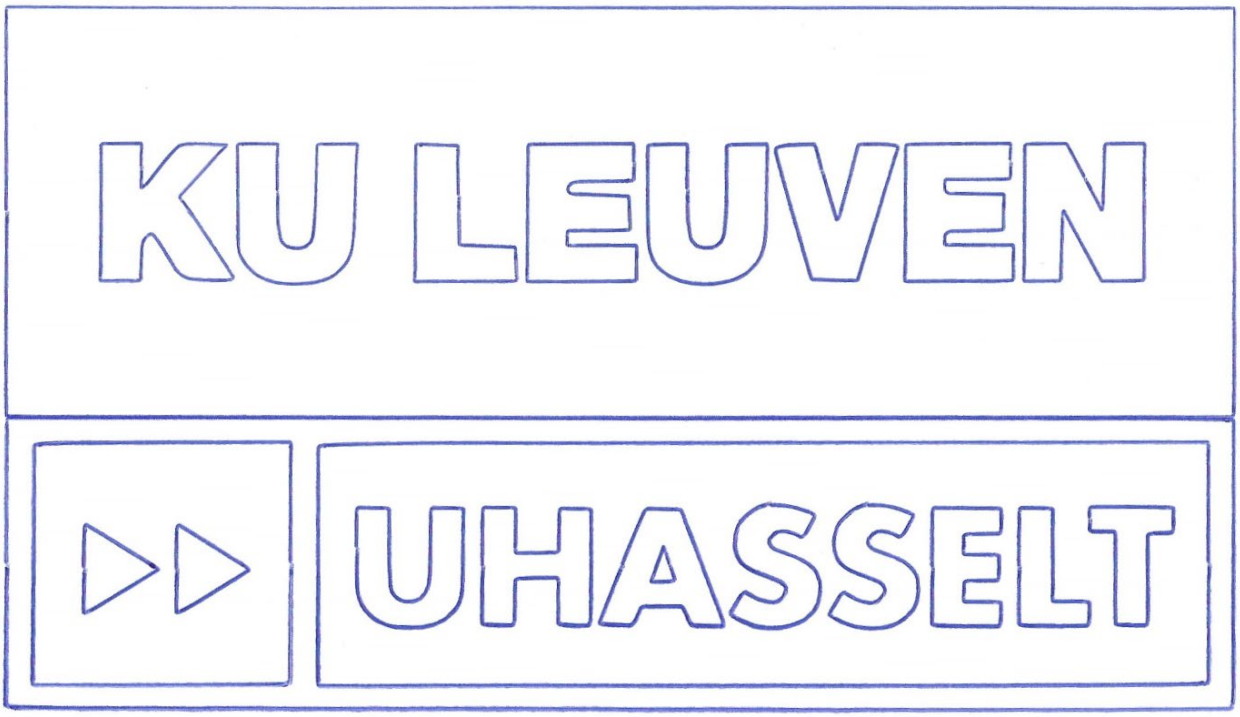
Figuur 44: KU Leuven en UHasselt logo lineair ongefilterd

Voor de volgende testen worden de punten telkens gefilterd met verschillende parameters. Voor de eerste test zijn de parameters van de algoritmes als volgt: de tolerantie voor het Douglas-Peucker-algoritme is 1 mm, de hoek- en oriëntatietolerantie zijn beiden 140 graden. Voor het filteren bestond het bestand uit 11804 punten en na het filter met deze parameters nog slechts 616 punten. Figuur 45 geeft het resultaat weer, hieruit blijkt dat kromme paden niet goed getekend worden. Dit komt doordat de tolerantie van het Douglas-Peucker-algoritme te hoog stond. De robot deed er 28 min en 32 seconden over om beide logo's te tekenen. Dit resulteert in een snelheidsverbetering van 74%.



Figuur 45: KU Leuven en UHasselt logo, lineair, gefilterd: DPT = 1mm, hoektolerantie = 140°, oriëntatietolerantie = 140°

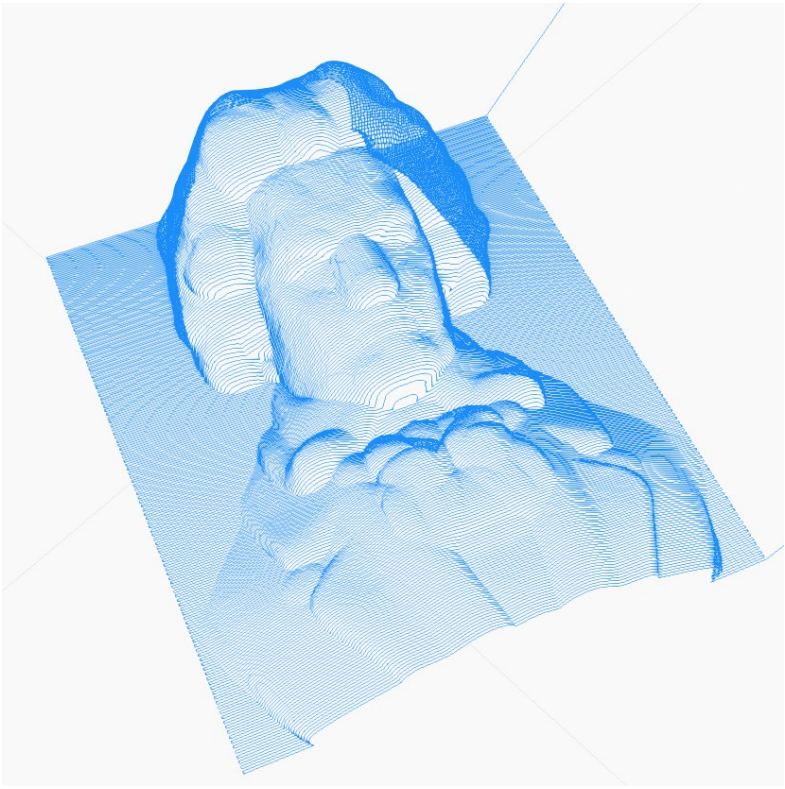
In Figuur 46 wordt de tolerantie van het Douglas-Peucker-algoritme verlaagd naar 0,1 mm, voor de rest worden dezelfde parameterinstellingen gebruikt als in de vorige test. Het aantal punten na het filteren is 754 punten. Uit Figuur 46 blijkt dat de kromme paden correct worden getekend. De tijd die nodig is om de logo's te tekenen is 30 minuten en 21 seconden, dit resulteert in een snelheidsverbetering van 71%.



Figuur 46: KU Leuven en UHasselt logo, lineair, gefilterd: DPT = 0,1mm, hoektolerantie = 140°, oriëntatietolerantie = 140°

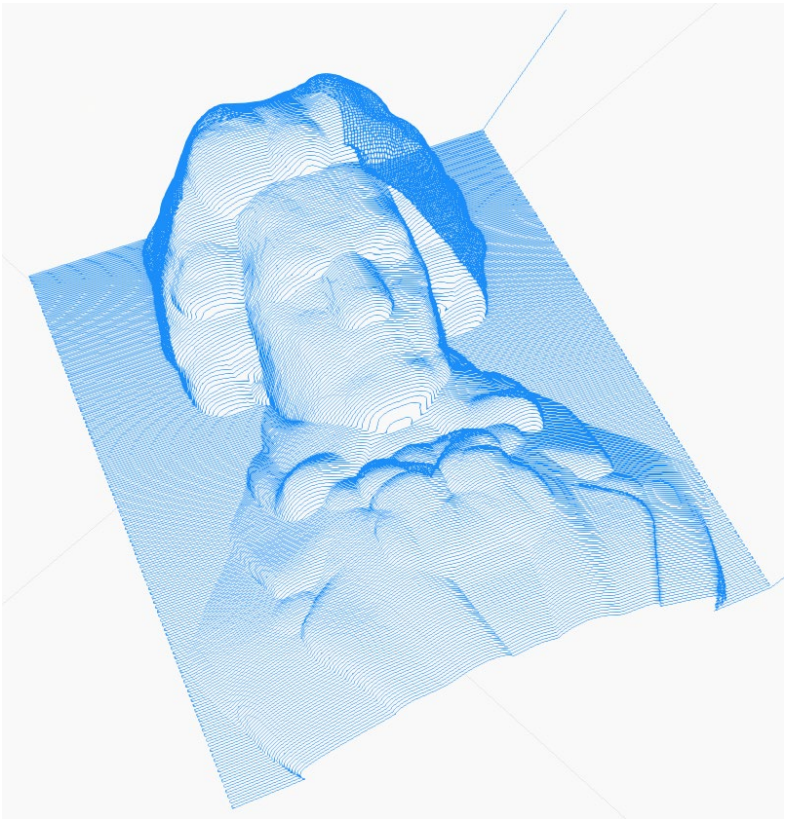
5.2 3D-testen

Nadat het programma in 2D getest werd, is de volgende stap om het programma te testen in 3D. Door gebruik te maken van *KUKA.Sim* kunnen CSV bestanden ingelezen worden en kan de beweging van de robot gesimuleerd worden. De testen zullen uitgevoerd worden op een bestaand bestand, dat het hoofd van Beethoven beschrijft. Voor de lineaire testen in 3D is de maximale snelheid van de eindeffector ingesteld op 0,0833 m/s. Het bestand bevat oorspronkelijk 28105 punten en de simulatietijd is gelijk aan 8 uur 39 min 15 sec. De werkelijke tijd is gelijk aan de simulatietijd plusminus een afwijking van 2% [18]. Figuur 47 geeft het gevolgde pad weer.



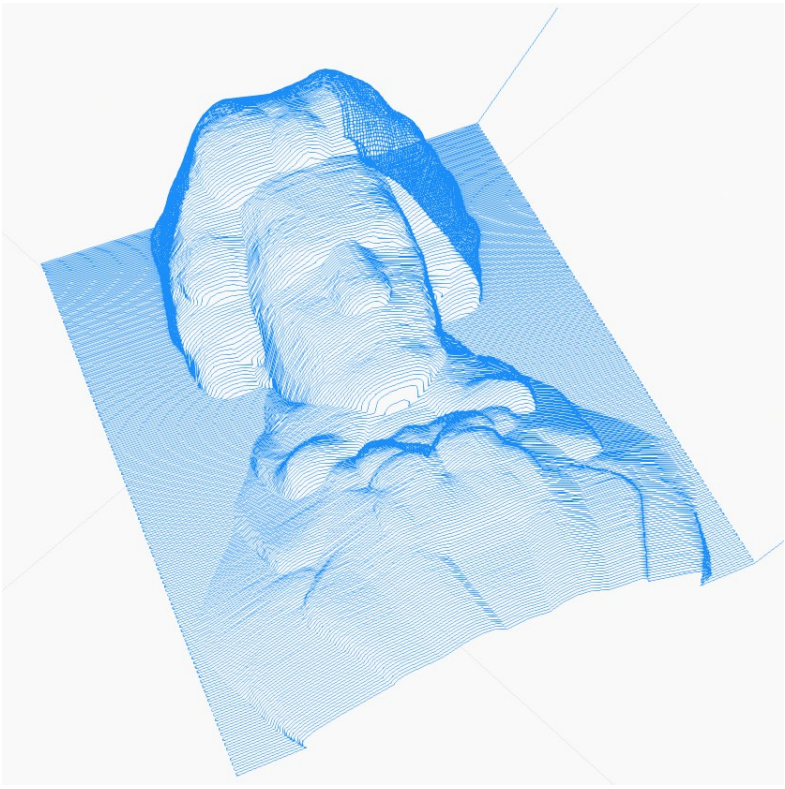
Figuur 47: Beethoven ongefilterd

Om het programma te testen, wordt het puntenbestand met twee verschillende toleranties voor het Douglas-Peucker-algoritme gefilterd. De andere toleranties zijn: 140° voor het hoekalgoritme en 140° voor het oriëntatiealgoritme. Deze toleranties blijven voor de twee testen constant. Als eerst wordt het bestand gefilterd met een tolerantie van 0,1 mm. Het aantal overgebleven punten is gelijk aan 17189 en de simulatietijd is 5 uur 13 min 59 sec. Hieruit volgt dat de snelheidsverbetering gelijk is aan 40%. Figuur 48 geeft het gevolgde pad weer.



Figuur 48: Beethoven gefilterd, lineair, D P= 0.1 mm, hoektol = 140° en oriëntatietol = 140°

Voor de tweede test wordt de tolerantie van het Douglas-Peucker-algoritme verhoogt naar 0,5 mm. Het aantal overgebleven punten is gelijk aan 9466 en de simulatie snelheid is 2 uur 51 min 1 sec. Hieruit volgt dat de snelheidsverbetering gelijk is aan 67%. Figuur 49 geeft het gevolgde pad weer.



Figuur 49: Beethoven gefilterd, lineair, DP = 0.5 mm, hoektol = 140° en oriëntatietol = 140°

Vervolgens wordt één van deze bestanden gefreesd door gebruik te maken van de robot. Het materiaal waarin gefreesd wordt is een zacht materiaal zodat onverwachte fouten geen ernstige gevolgen veroorzaken. Figuur 50 geeft het resultaat weer van de freestest waarbij de Douglas-Peucker-tolerantie gelijk is aan 0,1 mm.

Figuur 50: Freestest 1 Beethoven, lineair, DP = 0,1 mm, hoektol = 140°, oriëntatietol = 140°

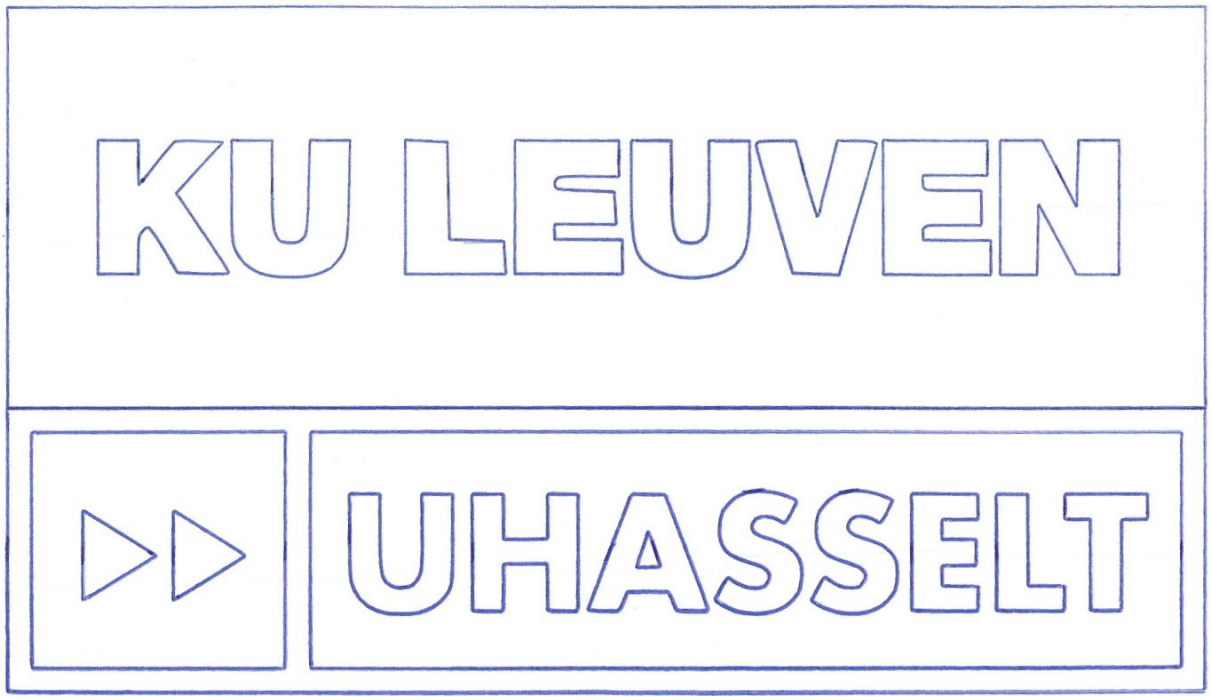
Uit de freestesten blijkt dat het programma correct werkt voor lineaire bewegingen. Ook is de snelheidsverbetering door het filteren van de punten aanzienlijk ongeacht de gekozen tolerantie. Indien de toleranties niet te groot gekozen worden, blijven ook alle details behouden.

6 Spline-beweging

6.1 2D-testen

Net zoals bij de lineaire testen worden ook de testen met een spline-beweging altijd in auto-modus uitgevoerd met een maximale snelheid van 2 m/s die verder gereduceerd wordt tot 10%.

Om spline-bewegingen te kunnen maken moeten de punten in spline blocks geplaatst worden. Het aantal punten per spline block is cruciaal voor de tijd die nodig is om het pad te berekenen. Allereerst wordt het ongefilterde bestand getest en worden alle 11804 punten in 1 spline block geplaatst. Figuur 51 geeft het resultaat hiervan weer. Omdat alle punten in 1 spline block staan, moet de robot eerst het pad berekenen. De robot deed hier 1 minuut en 54 seconden over. Het tekenen zelf duurde 11 minuten en 17 seconden. Echter waren er aan deze methode, waarin slechts 1 spline block gebruikt werd, ook nadelen. Zo neemt het berekenen veel werkgeheugen in beslag, hierdoor kan de robot *crashen*. Deze crash kan voor of na het uitvoeren van de spline block voorkomen.



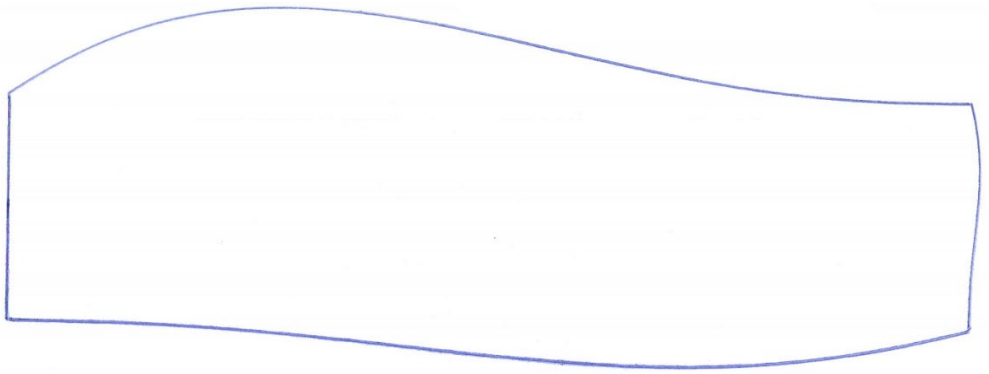
Figuur 51: KU Leuven en UHasselt logo spline, 1 block, ongefilterd

Hierna werden er nog testen uitgevoerd met 100 en 50 punten per spline block. Bij beide testen begon de robot meteen met tekenen. Dit komt omdat de aantal punten per spline block veel kleiner is, waardoor het berekenen van het pad sneller gebeurt. De tijd om de logo's te tekenen met 100 punten per spline block is 14 minuten en 23 seconden. Indien er zich 50 punten per spline block bevinden is de tijd 17 minuten en 39 seconden. Net zoals wanneer er slechts 1 grote spline block gebruikt wordt, is er bij het gebruik van 100 punten per spline block ook een kans dat de robot crasht. Deze crash zal enkel gebeuren nadat alle paden volledig uitgevoerd zijn, maar gebeurt niet elke keer. Door de mogelijkheid tot crashen wordt er in het vervolg van deze thesis enkel nog gebruik gemaakt van spline blocks met 50 punten per block.

Vervolgens worden de punten opnieuw gefilterd en getekend met behulp van spline-paden. In de eerste test worden de gefilterde punten getekend met als parameterinstellingen: Douglas-Peucker-algoritme is 1 mm, de hoek- en oriëntatietolerantie zijn beiden 140 graden. Figuur 52 geeft het resultaat weer. Hierin is te zien dat de robot de meeste lijnen niet volledig recht tekent maar in een boog naar het volgende punt gaat.

Figuur 52: KU Leuven en UHasselt logo, spline, gefilterd: DPT = 0,1 mm, hoektolerantie = 140°, oriëntatietolerantie = 140°

Echter is het merkwaardig dat het kader van het logo van de KU Leuven wel recht is en dat van de UHasselt niet. Om te verklaren waarom het kader van het logo van de UHasselt nier recht is, wordt er enkel gekeken naar de punten van het kader. Het filteralgoritme houdt voor en na elk hoekpunt een extra punt over, deze punten liggen ongeveer 0,2 mm van het hoekpunt. Voor het kader van het logo van de KU Leuven is deze afstand ongeveer hetzelfde. Deze kleine afstand tussen de punten is bij een lineaire beweging van de robot nodig zodat de hoeken niet afgerond worden. Bij een spline-beweging zorgt dit er echter ook voor dat de robot in het hoekpunt zeer traag is. Figuur 53 geeft het kader van het UHasselt logo weer.

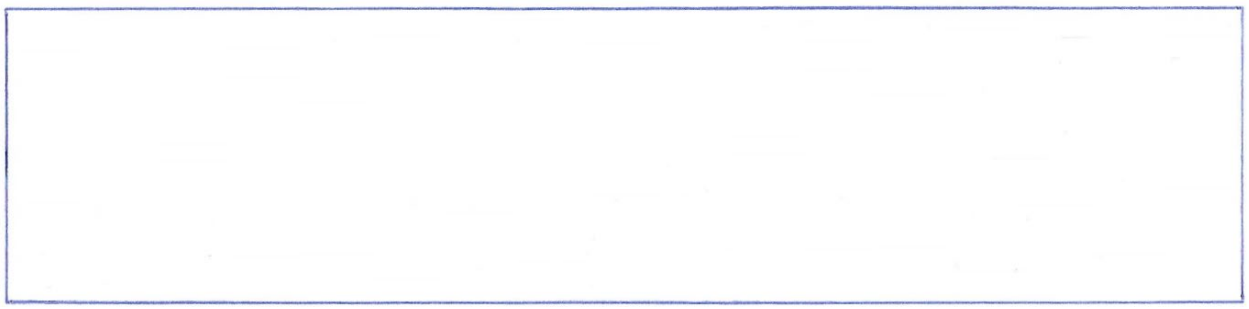


Figuur 53: Vierkant UHasselt logo, gefilterd: DPT = 0,1 mm, hoektolerantie = 140°, oriëntatietolerantie = 140°

Indien de punten voor en na de hoekpunten handmatig vervangen worden door andere punten die op een afstand van 0,2 mm van het hoekpunten liggen, wordt het kader wel recht getekend. Na verder onderzoek naar het probleem bleek dat het probleem lag bij het online programma dat de punten genereert. Doordat de punten bepaald worden aan de hand van een SVG-bestand is het mogelijk dat indien er een rechte getekend is, het programma de punten niet 100% op deze rechte plaatst. Verder worden de punten door het programma op een bepaalde afstand van elkaar geplaatst. Hierdoor is het zeer waarschijnlijk dat hoekpunten niet perfect in de hoek staan. Figuur 54 geeft het probleem weer, hierbij ligt één van de punten niet perfect op de rechte.

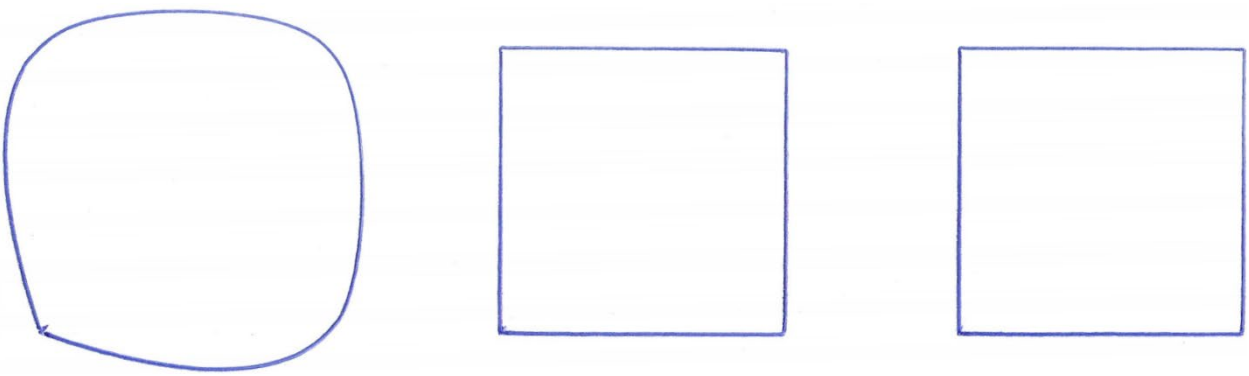
Figuur 54: Fout bij PathToPoints

Doordat het filteralgoritme de hoekpunten en het punt voor en na het hoekpunt bijhoudt en deze punten niet perfect op de juiste plaats liggen zal de spline-beweging door deze punten niet altijd recht zijn. Om te voorkomen dat de robot niet het gewenste pad volgt bij kleine fouten in de punten wordt het filteralgoritme aangepast. Het algoritme zal na deze aanpassingen zelf punten voor en na een hoek genereren die perfect op een rechte liggen met het vorige punt. De afstand tussen het hoekpunt en de toegevoegde punten wordt als 1 mm genomen. Bij deze waarden zal de robot steeds een rechte lijn tekenen indien nodig zonder af te ronden. De volledige werking van deze aanpassing van het hoekalgoritme zijn te vinden in hoofdstuk 4.8. Het resultaat van deze aanpassingen op het kader van het UHasselt logo is weergegeven in Figuur 55.



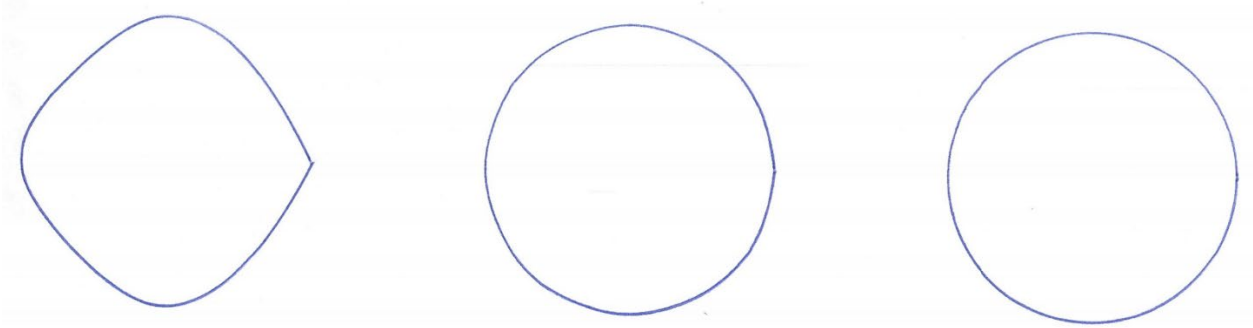
Figuur 55: Vierkant UHasselt logo, 3 punten in elke hoek

Verder worden er nog enkele simpele testen uitgevoerd om het nieuwe filteralgoritme uit te testen. In een eerste test worden er vierkanten getekend. In Figuur 56a werd een vierkant getekend met 5 punten, hiervoor had de robot 16 seconden nodig. In Figuur 56b wordt een vierkant met in elk hoekpunt drie punten getekend, hiervoor had de robot 32 seconden nodig. In Figuur 56c wordt een vierkant getekend met in elk hoekpunt drie punten en in het midden van elke zijde nog een punt, hiervoor had de robot 31 seconden nodig. De robot zal dus trager worden door punten in de hoeken toe te voegen, dit is logisch aangezien de robot zijn assen niet snel genoeg kan versnellen om deze hoeken te maken. Echter is er tussen Figuur 56b en Figuur 56c nauwelijks een verschil in snelheid. Hieruit kan geconcludeerd worden dat een punt toevoegen op een rechte lijn niet zorgt voor een tragere werking indien de robot in spline staat. Het gevolgde pad wordt er echter ook niet beter door.



Figuur 56: Vierkant, (a) 5 punten, (b) 3 punten per hoek, (c) 3 punten per hoek en extra punt

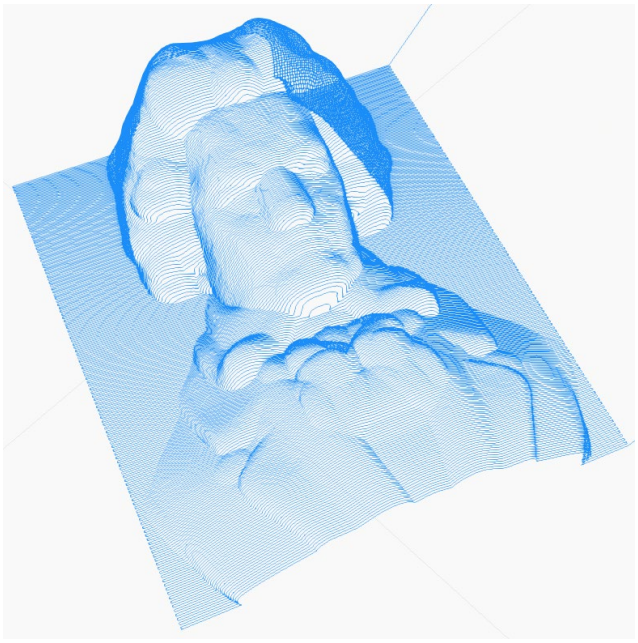
Een tweede test bestond eruit om te bestuderen hoeveel punten er nodig zijn om een cirkel te tekenen met spline. Allereerst werd een cirkel met 5 punten (om de 90 graden een punt) getekend, dit wordt weergegeven in Figuur 57a. Hieruit volgt dat 5 punten te weinig is om een volledige cirkel te tekenen. Vervolgens worden er om de 45 graden punten geplaatst, deze cirkel is weergegeven in Figuur 57b. Uit deze figuur blijkt dat negen punten net te weinig is om een perfecte cirkel te tekenen. Tot slot werd de cirkel getekend met 48 punten, zie Figuur 57c. Hieruit blijkt dat 48 punten genoeg is om een cirkel te tekenen met spline. De tijd die nodig was om de cirkels te tekenen bleef ongeveer constant, namelijk 13 seconden. Hieruit kan geconcludeerd worden dat meer punten in een cirkel niet zorgen voor een tragere werking indien de robot in spline staat.



Figuur 57: Cirkel, (a) 5 punten, (b) 9 punten, (c) 48 punten

6.2 3D-testen

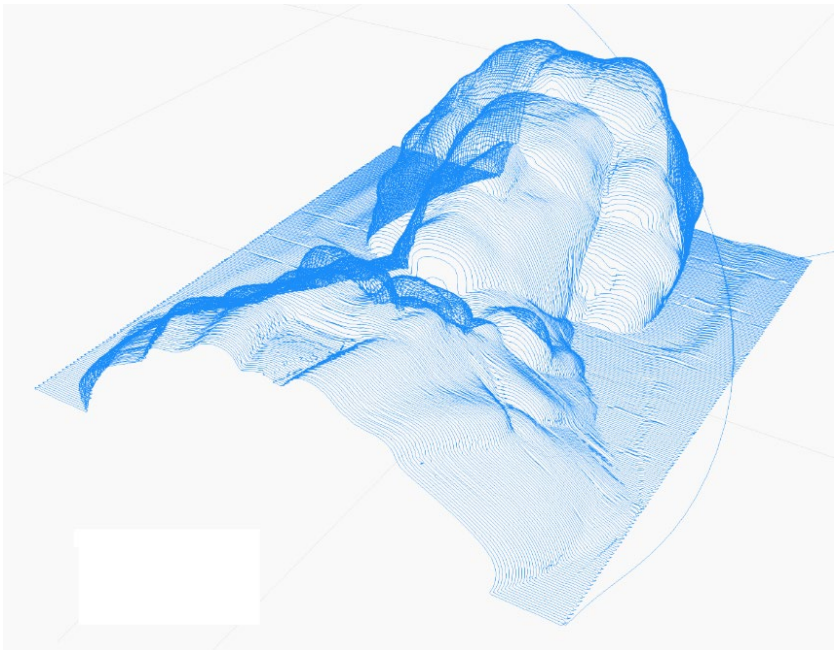
Net zoals bij de lineaire bewegingen, wordt het programma ook voor spline-bewegingen getest in 3D aan de hand van simulaties. Hiervoor wordt er weer gekozen voor het puntenbestand van Beethoven. Er wordt één test uitgevoerd waarbij de Douglas-Peucker tolerantie gelijk is aan 0,1 mm. De andere toleranties zijn opnieuw 140° voor het hoekalgoritme en 140° voor het oriëntatiealgoritme. Eerst wordt het bestand ongefilterd weergeven, waarbij de punten lineair met elkaar verbonden worden. De simulatietijd voor dit bestand is zoals eerder vermeld gelijk aan 8 uur 39 min 15 sec. Hierna kan het gefilterde pad met spline werking visueel vergeleken worden met zijn oorspronkelijke lineaire vorm. Figuur 58 geeft het ongefilterde pad weer.



Figuur 58: Beethoven ongefilterd

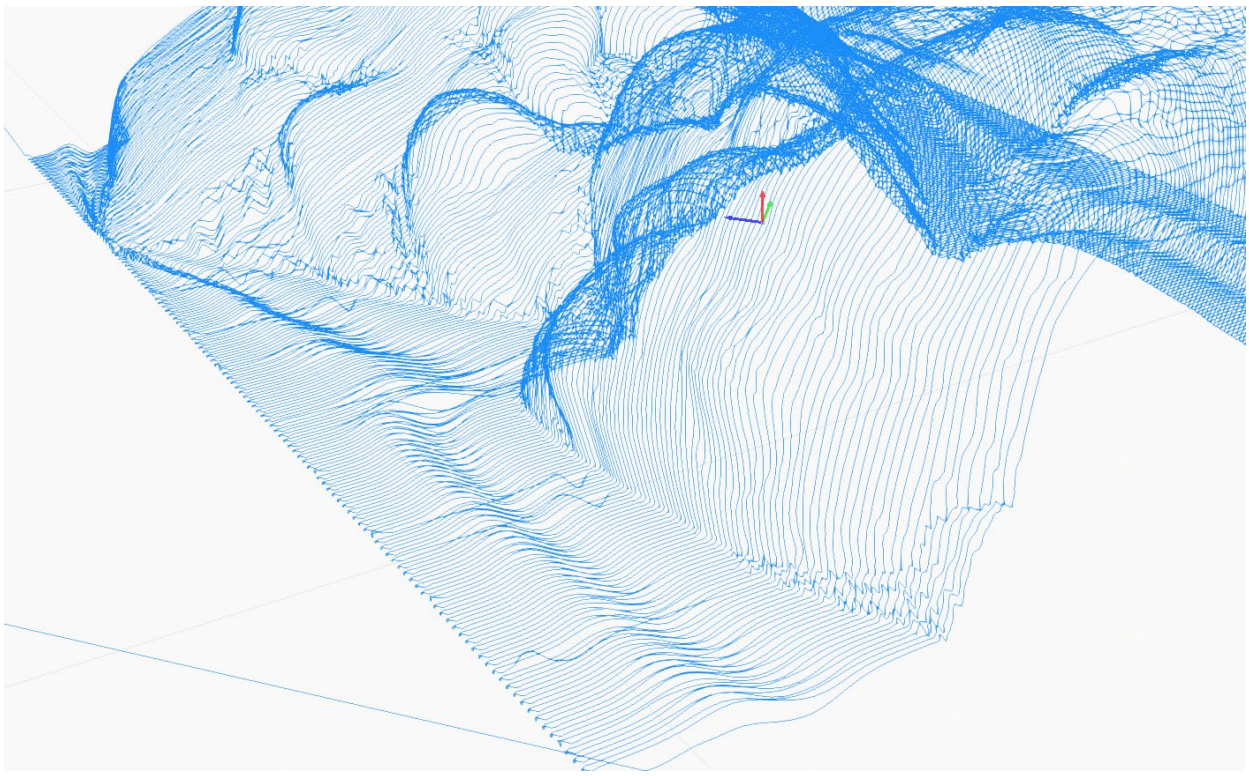
Vervolgens wordt dit bestand gefilterd met behulp van het programma. Het gefilterde bestand bevat 17790 punten, deze punten worden per 50 in één spline block geplaatst. Er zijn dus in totaal 356 spline blocks. Voor het simuleren van het pad wordt gebruik gemaakt van een maximale robotsnelheid van 2 m/s. Deze snelheid wordt verder gereduceerd naar 60% van zijn maximale snelheid, waardoor de totale simulatie tijd gelijk is aan 908.488 sec. Figuur 59 geeft het resultaat weer.

Om de snelheid van de spline-bewegingen te vergelijken met het ongefilterde bestand, moet de tijd omgerekend worden. Voor de spline-bewegingen wordt er gebruik gemaakt van een maximale robotsnelheid van 2 m/s die verder gereduceerd wordt tot 60% hiervan. De maximale robotsnelheid bij het originele puntenbestand met lineaire werking is 0,0833 m/s. De spline-beweging heeft dus een maximale ingestelde snelheid die 14,4 keer sneller is dan die van de lineaire bewegingen. Indien we de tijd van de spline simulatie vermenigvuldigen met 14,4 bekomen we een tijd van 3 uur 38 min 2 sec. Hieruit volgt dat de snelheidsverbetering gelijk is aan 58% ten opzichte van het ongefilterde bestand. Echter is dit niet volledig correct aangezien de maximale snelheden met elkaar vergeleken worden. Hierdoor is de snelheidsverbetering maar een schatting van de werkelijke snelheidsverbetering, maar deze schatting zal in de buurt van de werkelijke verbetering liggen.



Figuur 59: Gefilterd, spline, DP = 0,1 mm, hoektol = 140° en oriëntatietol = 140°

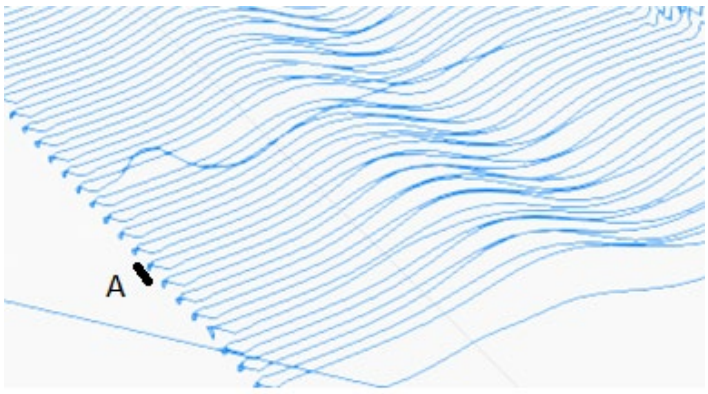
Op het eerste zicht lijkt het gefilterd bestand op het originele bestand. Echter zitten er kleine fouten in. Ten eerste zijn er golven aan de linker en rechter kant, dit zouden normaal rechte stukken moeten zijn. Ten tweede zitten er verschillende kleine pieken in het gefilterde pad. Figuur 60 geeft een beter beeld weer van de twee fouten.



Figuur 60: Vergroting van fouten bij spline test

De fouten kunnen verminderd worden door de snelheid van de robot te verlagen. Het pad bij spline-bewegingen is namelijk afhankelijk van de gekozen snelheid. Indien de snelheid te hoog is, zal het pad veranderen en zullen er grotere afwijkingen ontstaan. Echter zal hierdoor het probleem niet verholpen zijn. Een mogelijke oplossing voor de golvende paden is één of meerdere punten toevoegen op de rechte lijn. Voor het verminderen van de uitschieters in het pad kunnen opnieuw extra punten toegevoegd worden aan het pad zodat het pad nauwkeuriger wordt.

Indien het linkerdeel van de figuur verder vergroot wordt, dan valt er een derde fout op. Figuur *61* geeft deze fout weer. Het uiteinde van het pad aan de linker- en rechterkant maakt geen hoek van 90°. Dit komt doordat de afstand *A* tussen twee verschillende lijnen kleiner is dan 2 mm. Hierdoor kan er geen punt toegevoegd worden voor en achter de hoekpunten. Een mogelijke oplossing voor deze fout is het variabel maken van de afstand zodat ook wanneer de afstand kleiner is dan 2 mm, er toch een punt voor of na het hoekpunt toegevoegd kan worden.



Figuur 61*: Fout bij opeenvolgende hoeken met een kleine afstand A*

Zoals eerder gezegd, probeert de robot bij spline-bewegingen zijn snelheid constant te houden. Deze snelheid wordt enkel gelimiteerd door de jerk. Tijdens bovenstaande simulatie werd de jerk maximaal

gekozen. Hierdoor maakt de robot op sommige momenten schokkende bewegingen. Dit kan trillingen in de eindeffector veroorzaken waardoor het pad afwijkt van het lineaire pad. De jerk moet dus gelimiteerd worden om dit te voorkomen.

Om een vergelijking te maken tussen de simulatietijd met lineaire en spline-bewegingen wordt opnieuw de tijd van de spline-simulatie vermenigvuldigd met 14,4. Dit geeft een totale tijd van 3 uur 38 min 2 sec. De totale tijd voor de simulatie met lineaire bewegingen en een tolerantie van 0,1 mm was 5 uur 13 min 59 sec. Hieruit volgt dat de simulatie met spline-bewegingen 1 uur 35 min 57 sec of 31,6% sneller is dan de simulatie met lineaire bewegingen. Echter geven lineaire bewegingen een veel nauwkeuriger resultaat.

7 Besluit

De hoofddoelstelling van dit onderzoek is om het freesproces te versnellen met behoud van de nauwkeurigheid binnen een gegeven tolerantie voor zowel lineaire bewegingen als voor spline-bewegingen. Dit gebeurt door punten in het robotpad te verwijderen en toe te voegen door middel van een algoritme. Uit een voorstudie is gebleken dat het Douglas-Peucker-algoritme het meest geschikt is voor deze toepassing, omdat het algoritme het nauwkeurigste is. Echter is dit wel het traagste algoritme, maar uit testen bleek dat het algoritme één miljoen punten in minder dan één seconde filtert.

Uit de 2D testen waarbij de robot lineaire bewegingen met luswerking maakt, blijkt dat er nog met extra factoren rekening gehouden moet worden zoals de hoek tussen de punten, oriëntatie van de eindeffector en de draairichting van de spindel. Vervolgens bleek uit de testen dat er grote snelheidsverbeteringen gerealiseerd kunnen worden door de puntenbestanden op voorhand te filteren. Tijdens de testen werden snelheidsverbeteringen tot 74% gerealiseerd, waarbij er visueel geen nauwkeurigheidsverlies waar te nemen was. Uit de 2D testen waarbij de robot spline-bewegingen maakt, blijkt dat het algoritme voor de lineaire bewegingen aangepast moet worden. Voor spline-bewegingen moeten er echter punten toegevoegd worden. Ook bleek dat 50 punten per spline block ideaal is om stilstanden te voorkomen.

Tijdens 3D simulaties met lineaire bewegingen zijn er snelheidsverbeteringen tot 40% gemeten, waarbij de tolerantie van het Douglas-Peucker-algoritme gelijk is aan 0,1 mm. Indien de tolerantie verhoogd werd naar 0,5 mm, werden snelheidsverbeteringen tot 67% gemeten. Uit de 3D simulaties met spline-bewegingen blijkt dat het algoritme niet volledig correct werkt. De globale vorm van de gefilterde objecten blijft goed behouden. Echter ontstaan er kleine afwijkingen in het pad zoals golvende paden die normaal recht zouden moeten zijn en onverwachte uitschieters in het pad. Spline-bewegingen zijn echter wel sneller dan lineaire bewegingen. Uit testen bleek dat er een snelheidsverbetering tot 31,6% mogelijk was ten opzichte van lineaire bewegingen.

Voor het filteren van spline-bewegingen zijn nog verbeteringen mogelijk om deze afwijkingen weg te werken. Doordat het pad bij spline-bewegingen afhankelijk is van de snelheid van de robot is het echter moeilijk om voor elke snelheid de juiste filterinstellingen te bepalen. Ook zullen er grote schokken in de robot ontstaan indien de robot van richting moet veranderen waardoor de jerk best gelimiteerd wordt. Er is dus nog verder onderzoek nodig om punten te filteren voor deze bewegingen.

Toekomstig werk

Het huidige filterprogramma is een goede basis voor het filteren van punten uit een robotpad waardoor de robot het freesproces sneller kan uitvoeren. Indien KUKA het programma voor iedere toepassing wil gebruiken zijn er echter nog enkele aanpassingen nodig. Zo is het belangrijk dat er naast CSV bestanden ook BIN en TXT files ingelezen kunnen worden. Ook moet het mogelijk gemaakt worden dat andere parameterindelingen van het puntenbestand ingelezen kunnen worden.

Specifiek voor de werking met spline-bewegingen moeten er nog enkele aanpassingen gebeuren aan het filterprogramma. Om te beginnen moet de robotcode aangepast worden zodat het CAMrob softwarepakket van KUKA ook deze bewegingen ondersteunt. Ook moet er onderzocht worden of het filteralgoritme verder geoptimaliseerd kan worden voor spline-bewegingen.

Een laatste mogelijke verbetering is het filterprogramma online laten werken. Hiervoor moet het programma geïntegreerd worden in het CAMrob softwarepakket. De robot kan dan tijdens het frezen de punten filteren. Hierdoor wordt er geen tijd verloren met op voorhand punten te filteren. Indien er beslist wordt om het filterprogramma niet online te laten werken, dan kan er in de plaats een GUI gemaakt worden zodat het programma eenvoudig gebruikt kan worden.

Literatuurlijst

[1] KUKA." Frezen." https://www.kuka.com/nl-be/producten diensten/procestechnologie%C3%ABn/2016/07/frezen (geopend 11 oktober, 2019).

[2] A. Owen-Hill. "Can a Robot Outperform a CNC Machine for Robot Machining?" https://robodk.com/blog/robot-machining-vs-cnc/ (geopend 29 oktober, 2019).

[3] Metaradam. "Industrial Robot Machine KUKA Milling." https://www.hiclipart.com/free-transparent-background-png-clipart-mhjwn (geopend 12 oktober, 2019).

[4] Ş. Ekdemir, "Efficient implementation of polyline simplification for large datasets and usability evaluation," [eindwerk], Uppsala: Department of Information Technology, 2011.

[5] M. Visvalingam, "The Visvalingam algorithm: metrics, measures and heuristics," The Cartographic Journal, vol. 53, no. 3, pp. 242-252, 2016.

[6] M. Visvalingam and P. J. Williamson, "Simplification and generalization of large scale data for roads: a comparison of two filtering algorithms," *Cartography and Geographic Information Systems*, vol. 22, no. 4, pp. 264-275, 1995.

[7] Gettysburg. "Introduction to Scientific Computation." http://www.cs.gettysburg.edu/~ilinkin/courses/Spring-2016/cs107/assignments/a6.html (geopend 25 november, 2019).

[8] W. Shi and C. Cheung, "Performance evaluation of line simplification algorithms for vector generalization," *The Cartographic Journal*, vol. 43, no. 1, pp. 27-44, 2006.

[9] Y.-h. Liu and W.-q. Chen, "Line simplification algorithm implementation and error analysis," in *2011 IEEE International Conference on Computer Science and Automation Engineering*, 2011, vol. 2, pp. 64-68: IEEE.

[10] S.-T. Wu, A. C. da Silva, and M. R. Márquez, "The Douglas-peucker algorithm: sufficiency conditions for non-self-intersections," *Journal of the Brazilian Computer Society*, vol. 9, no. 3, pp. 67-84, 2004.

[11] H. Nie and Z. Huang, "A new method of line feature generalization based on shape characteristic analysis," *Metrology and Measurement Systems*, vol. 18, no. 4, pp. 597-606, 2011.

[12] KUKA, *Robotprogrammering 1*. 2013, p. 257.

[13] E. Demeester and J. Baeten, "Robotics - manipulators Spatial transformations, kinematics, Jacobians,trajectory generation, construction, programming, applications," [curus], Diepenbeek: Gezamenlijke opleiding Industriële Ingenieurswetenschappen UHasselt & KU Leuven, 2019.

[14] KUKA, "KUKA.CAMRob KRC V3.0," p. 41, 2017.

[15] KUKA, "KUKA System Software," vol. 7, p. 639, 2019.

[16] L. L. Schumaker, *Spline functions : basic theory*. New York (N.Y.) : Wiley, 1981.

[17] E. d. Koning. "Douglas-Peucker simplification." http://psimpl."sourceforge.net/douglas-peucker.html (geopend 20 mei, 2020).

[18] M. Sools, "Measurements and evaluation of cycle times," p. 8, 2019.

Bijlagen

Bijlage A: Punten inlezen

```
public static int MakePointList(List<Point> InputPoints, string path)
        {
            StreamReader reader = new StreamReader(File.OpenRead(path));
            string line = reader.ReadLine();
            int counter = 1;
            while (line != null)
            {
                string[] values = line.Split(';');

                InputPoints.Add(new Point(
                    counter,
                    double.Parse(values[1], CultureInfo.InvariantCulture),
                    double.Parse(values[2], CultureInfo.InvariantCulture),
                    double.Parse(values[3], CultureInfo.InvariantCulture),
                    double.Parse(values[4], CultureInfo.InvariantCulture),
                    double.Parse(values[5], CultureInfo.InvariantCulture),
                    double.Parse(values[6], CultureInfo.InvariantCulture),
                    counter,
                    double.Parse(values[8], CultureInfo.InvariantCulture),
                    int.Parse(values[9], CultureInfo.InvariantCulture),
                    int.Parse(values[10], CultureInfo.InvariantCulture),
                    int.Parse(values[11], CultureInfo.InvariantCulture),
                    int.Parse(values[12], CultureInfo.InvariantCulture),
                    int.Parse(values[13], CultureInfo.InvariantCulture),
                    double.Parse(values[14], CultureInfo.InvariantCulture),
                    int.Parse(values[15], CultureInfo.InvariantCulture),
                    int.Parse(values[16], CultureInfo.InvariantCulture),
                    int.Parse(values[17], CultureInfo.InvariantCulture),
                    int.Parse(values[18], CultureInfo.InvariantCulture),
                    int.Parse(values[19], CultureInfo.InvariantCulture),
                    int.Parse(values[20], CultureInfo.InvariantCulture)));
                line = reader.ReadLine();
                counter++;
            }
            return InputPoints.Count;
        }
```

Bijlage B: Douglas-Peucker-algoritme

```
public BitArray FilterDouglasPeucker(List<Point> points, double tolerance)
        {
            BitArray bitArray = new BitArray(points.Count);
            bitArray.Set(0, true);
            bitArray.Set(points.Count - 1, true);
            Stack<Range> stack = new Stack<Range>();
            stack.Push(new Range(0, points.Count - 1));

            while (stack.Count > 0)
            {
                Range range = stack.Pop();

                int index = -1;
                double maxDist = 0f;

                if (Distance.GetDistance(points[range.First],
                                         points[range.Last]) == 0)
                {
                    for (int i = range.First + 1; i < range.Last; ++i)
                    {
                        double dist = Distance.GetDistance(
                        points[i], points[range.First]);
                        if (dist > maxDist)
                        {
                            index = i;
                            maxDist = dist;
                        }
                    }

                    if (maxDist > tolerance)
                    {
                        bitArray.Set(index, true);

                        stack.Push(new Range(range.First, index));
                        stack.Push(new Range(index, range.Last));
                    }
                }
                else
                {
                    for (int i = range.First + 1; i < range.Last; ++i)
                    {
                        double dist = Distance.GetSegmentDistance(
                        points[i], points[range.First], points[range.Last]);
                        if (dist > maxDist)
                        {
                            index = i;
                            maxDist = dist;
                        }
                    }

                    if (maxDist > tolerance)
                    {
                        bitArray.Set(index, true);

                        stack.Push(new Range(range.First, index));
                        stack.Push(new Range(index, range.Last));
                    }
                }
            }
            return bitArray;
        }
```

Bijlage C: Hoekalgoritme

```
public BitArray FilterCorner(List<Point> points, double cornerTol)
        {
            BitArray bitArray = new BitArray(points.Count);
            bitArray.Set(0, true);
            bitArray.Set(points.Count - 1, true);
            Point p0, p1, p2, p3;
            double angle, pangle = 0, nangle;
            double cosphi;
            double[] v, w;

            p0 = points[0];
            p1 = points[1];
            p2 = points[2];

            v = new double[] { p1.X - p0.X, p1.Y - p0.Y, p1.Z - p0.Z };
            w = new double[] { p2.X - p1.X, p2.Y - p1.Y, p2.Z - p1.Z };
            cosphi = (v[0] * w[0] + v[1] * w[1] + v[2] * w[2]) / (Math.Sqrt(v[0]*v[0]
                      + v[1]*v[1] + v[2]*v[2]) * Math.Sqrt(w[0]*w[0] + w[1]*w[1] +
                      w[2]*w[2]));
            angle = 180 - (Math.Acos(cosphi) * 180) / Math.PI;

            for (int i = 1; i < points.Count - 2; ++i)
            {
                p1 = points[i];
                p2 = points[i + 1];
                p3 = points[i + 2];

                v = new double[] { p2.X - p1.X, p2.Y - p1.Y, p2.Z - p1.Z };
                w = new double[] { p3.X - p2.X, p3.Y - p2.Y, p3.Z - p2.Z };

                cosphi = (v[0] * w[0] + v[1] * w[1] + v[2] * w[2]) /
                         (Math.Sqrt(v[0]*v[0] + v[1]*v[1] + v[2]*v[2]) *
                          Math.Sqrt(w[0]*w[0] + w[1]*w[1] + w[2]*w[2]));
                nangle = 180 - (Math.Acos(cosphi) * 180) / Math.PI;

                if (angle > cornerTol)
                {
                    if (pangle < cornerTol)
                    {
                        if (nangle > cornerTol)
                        {
                            // point is the first point of a curve
                        }
                        else
                        {
                            // point is a vertex greater than the cornerTol
                            bitArray.Set(i - 1, true);
                            bitArray.Set(i, true);
                            bitArray.Set(i + 1, true);
                        }
                    }
                    else
                    {
                        // point is part of a curve
                        // or point is the last point of a curve
                    }
                }
```

```
            else
            {
                // point is a vertex less than the cornerTol
                bitArray.Set(i - 1, true);
                bitArray.Set(i, true);
                bitArray.Set(i + 1, true);
            }
            pangle = angle;
            angle = nangle;
        }
        return bitArray;
    }
```

Bijlage D: Oriëntatiealgoritme

```
public BitArray FilterOrientationTolerance(List<Point> points, double OrientationTol-
erance)
        {
            BitArray bitArray = new BitArray(points.Count);
            bitArray.Set(0, true);
            bitArray.Set(points.Count - 1, true);
            Point point = points[0];

            for (int i = 1; i < points.Count - 1; i++)
            {
                if (Math.Abs((Math.Abs(point.A) - Math.Abs(points[i].A))) >=
                            OrientationTolerance ||
                    Math.Abs((Math.Abs(point.B) - Math.Abs(points[i].B))) >=
                            OrientationTolerance ||
                    Math.Abs((Math.Abs(point.C) - Math.Abs(points[i].C))) >=
                            OrientationTolerance)
                {
                    bitArray.Set(i, true);
                    bitArray.Set(i + 1, true);


                    point = points[i];
                }
            }
            return bitArray;
        }
```

Bijlage E: Spindelalgoritme

```
public BitArray FilterSpindleState(List<Point> points)
        {
            BitArray bitArray = new BitArray(points.Count);
            bitArray.Set(0, true);
            bitArray.Set(points.Count - 1, true);
            Point p0;
            Point p1;

            for (int i = 1; i < points.Count - 1; i++)
            {
                p0 = points[i - 1];
                p1 = points[i];
                if (p0.Spindel != p1.Spindel || p0.Rpm != p1.Rpm ||
                    p0.RotDirection != p1.RotDirection)
                {
                    bitArray.Set(i, true);
                }
            }

            return bitArray;
        }
```

Bijlage F: Tijdsbepaling

```
public double TimeCalculator(List<Point> points)
        {
            double totalTime = 0;
            double totalDistance = 0;

            List<double> position = new List<double>();
            List<double> velocity = new List<double>();
            List<double> acceleration = new List<double>();

            position.Add(0);
            velocity.Add(0);
            acceleration.Add(0);

            for (int i = 1; i < points.Count; i++)
            {
                double distance = Distance.GetDistance(points[i - 1], points[i]);
                double maxVelocity = points[i].Feedrate * 1000;

                double tempDistance = (maxVelocity*maxVelocity) /
                                      (2 * maxAcceleration);
                if (distance != 0)
                {
                    if (tempDistance < distance / 2)
                    {
                        position.Add(totalDistance + tempDistance);
                        position.Add(totalDistance + distance - tempDistance);
                        velocity.Add(maxVelocity);
                        velocity.Add(maxVelocity);

                        totalDistance += distance;
                    }
                    else
                    {
                        position.Add(totalDistance + (distance / 2));
                        double velocityI = Math.Sqrt(2 * maxAcceleration *
                                                     (distance / 2));
                        velocity.Add(velocityI);
                        totalDistance += distance;
                    }
                }
            }

            for (int i = 1; i < position.Count; i++)
            {
                double accelerationI = (velocity[i]*velocity[i] -
                                        velocity[i - 1]*velocity[i - 1]) /
                                       (2 * (position[i] - position[i - 1]));
                acceleration.Add(accelerationI);
            }

            position.Add(totalDistance);
            velocity.Add(0);
            acceleration.Add(-maxAcceleration);

            for (int i = 1; i < position.Count; i++)
            {
                double time;
                if (velocity[i] == velocity[i - 1])
                {
                    time = (position[i] - position[i - 1]) / velocity[i];
                    totalTime += time;
                }
```

```
            else
            {
                time = (velocity[i] - velocity[i - 1]) / acceleration[i];
                totalTime += time;
            }

        }
        return totalTime;
    }
```

Bijlage G: Punten uitschrijven

```
public static int CreateCSV(List<Point> FilteredPoints, string path)
        {
            string fileName = Path.GetFileName(path);
            string filteredFile = "filtered_" + fileName;
            var csv = new StringBuilder();
            string filePath = $@"{Path.GetDirectoryName(path)}\{filteredFile}";
            filePath.Trim();
            System.Globalization.CultureInfo IC =
                System.Globalization.CultureInfo.InvariantCulture;
            for (int i = 0; i < FilteredPoints.Count; i++)
            {
                Point point = FilteredPoints[i];
                var line = string.Format("{0};{1};{2};{3};{4};{5};{6};{7};{8};
                            {9};{10};{11};{12};{13};{14};{15};{16};{17};{18};
                            {19};{20}",(i+1).ToString(IC), point.X.ToString(IC),
                            point.Y.ToString(IC), point.Z.ToString(IC),
                            point.A.ToString(IC), point.B.ToString(IC),
                            point.C.ToString(IC), point.OldNumber.ToString(IC)
                            point.Feedrate.ToString(IC), point.Spindel.ToString(IC),
                            point.Rpm.ToString(IC), point.RotDirection.ToString(IC),
                            point.Coolant.ToString(IC), point.ToolNo.ToString(IC),
                            point.Acceleration.ToString(IC), point.E1.ToString(IC),
                            point.E2.ToString(IC), point.E3.ToString(IC),
                            point.E4.ToString(IC), point.E5.ToString(IC),
                            point.E6.ToString(IC), Environment.NewLine);
                csv.AppendLine(line);
            }
            File.WriteAllText(filePath, csv.ToString());

            return FilteredPoints.Count;
        }
```

Bijlage H: Aanpassing hoekalgoritme voor spline

```
public List<Point> FilterCornerSpline(List<Point> points, double cornerTol)
        {
            int amount = points.Count;
            double angle;
            double pangle = 0;
            double nangle;
            int counter = 0;
            Point p0, p1, p2, p3;
            double cosphi;
            double[] v, w;

            p0 = points[0];
            p1 = points[1];
            p2 = points[2];

            v = new double[] { p1.X - p0.X, p1.Y - p0.Y, p1.Z - p0.Z };
            w = new double[] { p2.X - p1.X, p2.Y - p1.Y, p2.Z - p1.Z };
            cosphi = (v[0] * w[0] + v[1] * w[1] + v[2] * w[2]) / (Math.Sqrt(v[0]*v[0]
                      + v[1]*v[1] + v[2]*v[2]) * Math.Sqrt(w[0]*w[0] + w[1]*w[1] +
                      w[2]*w[2]));
            angle = 180 - (Math.Acos(cosphi) * 180) / Math.PI;

            for (int i = 1; i < amount - 2; ++i)
            {
                p1 = points[i + counter];
                p2 = points[i + 1 + counter];
                p3 = points[i + 2 + counter];

                v = new double[] { p2.X - p1.X, p2.Y - p1.Y, p2.Z - p1.Z };
                w = new double[] { p3.X - p2.X, p3.Y - p2.Y, p3.Z - p2.Z };

                cosphi = (v[0] * w[0] + v[1] * w[1] + v[2] * w[2]) /
                         (Math.Sqrt(v[0]*v[0] + v[1]*v[1] + v[2]*v[2]) *
                          Math.Sqrt(w[0]*w[0] + w[1]*w[1] + w[2]*w[2]));
                nangle = 180 - (Math.Acos(cosphi) * 180) / Math.PI;

                if (angle > cornerTol)
                {
                    if (pangle > cornerTol)
                    {
                        if (nangle > cornerTol)
                        {
                            // point is part of a curve
                        }
                        else
                        {
                            // point is the last point of a curve
                            counter = AddAfter(points, i + counter, counter);
                        }
                    }
                    else
                    {
                        if (nangle > cornerTol)
                        {
                            // point is the first point of a curve
                            counter = AddBefore(points, i + counter, counter);
                        }
```

```
                else
                {
                    // point is a vertex greater than the cornerTol
                    counter = AddBefore(points, i + counter, counter);
                    counter = AddAfter(points, i + counter, counter);
                }
            }
        }
        else
        {
            // point is a vertex smaller than the cornerTol
            counter = AddBefore(points, i + counter, counter);
            counter = AddAfter(points, i + counter, counter);
        }
        pangle = angle;
        angle = nangle;
    }
    return points;
}
```